Universidade de Aveiro Departamento de Electrónica e Telecomunicações
2010

Paulo Miguel Madureira Martins

Comunicações e armazenamento de massa em sistemas embebidos escaláveis

Communications and mass storage in scalable embedded systems

Universidade de Aveiro Departamento de Electrónica e Telecomunicações
2010

Paulo Miguel Madureira Martins

Comunicações e armazenamento de massa em sistemas embebidos escaláveis

Communications and mass storage in scalable embedded systems

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações (M.I.E.E.T.), realizada sob a orientação científica do Prof. Dr. Manuel Bernardo Salvador Cunha e do Prof. Dr. José Luis Costa Pinto Azevedo, Professores Auxiliares do Departamento de Electrónica Telecomunicações e Informática da Universidade de Aveiro

Dedico este trabalho à minha namorada e filho pelo incansável apoio, e ao Pedro Kulzer por acreditar na capacidade do grupo com quem trabalhei e por ter tido uma atitude sempre positiva face às adversidades que encontramos.
Não poderia deixar de agradecer também ao Prof. Doutor Bernardo Cunha pela ajuda prestada no desenvolvimento de soluções e conceitos no decorrer do projecto, cuja contribuição foi fundamental para ultrapassar questões de natureza ténica e cinetífica.
Gostaria de deixar o meu agradecimento ao grupo de trabalho, pelo excelente ambiente e camaradagem que se verificou durante o decorrer do projecto, proporcionando a melhor experiência de trabalho até à presente data.

o júri

Presidente	Prof. Doutor António Manuel Brito Ferrari Almeida Professor Catedrático da Universidade de Aveiro
Arguente	Prof. Doutor Paulo Francisco da Silva Cardoso Professor Auxiliar no Departamento de Electrónica Industrial da Universidade do Minho
Orientador	Prof. Doutor Manuel Bernardo Salvador Cunha Professor Auxiliar no Departamento do DETI
Co-orientador	Prof. Doutor José Luís Costa Pinto de Azevedo Professor Auxiliar no Departamento do DETI

palavras-chave

Sistemas Embebidos, Sistemas Integrados, Armazenamento de Massa, Comunicações, USB, FPGA, Xilinx, Spartan 3E

resumo

Inserido no projecto ECU2010, este documento visa determinar a melhor solução possível para implementação de armazenamento de informação e comunicações de elevado débito para aplicações no âmbito do desporto automóvel.
O projecto ECU2010 tem como objectivo a pesquisa de uma nova arquitectura de unidades de controlo electrónico (ECU) para desporto automóvel especialmente centrado no controlo de motores de combustão interna.
A nova arquitectura proposta deverá de ser capaz de fazer o controlo de um motor de combustão interna usando os mais modernos modelos de controlo, mas sendo baseada numa modelo de processamento distribuído, composta por módulos de processamento auto-suficientes ao nível de comunicações e armazenamento e de sensores/actuadores com inteligência capazes de processamento prévio de dados.
A comunicação entre módulos não será abordada neste documento nem a comunicação com os elementos periféricos de actuação e/ou natureza sensorial, mas sim a comunicação entre os módulos de processamento e um dispositivo de controlo e monitorização, doravante chamado de Anfitrião, que tipicamente será um computador pessoal ou PDA.
De igual forma este documento debruçar-se-á sobre uma solução para o armazenamento em massa de informação, principalmente focada no armazenamento de dados históricos resultantes de variáveis de monitorização, processamento intermédio e de actuação.
O objectivo deste documento será produzir um conjunto de blocos de electrónica digital reconfiguráveis implementando as funcionalidades atrás mencionadas numa FPGA da Xilinx modelo Spartan 3E, que em conjunto com hardware desenvolvido para o efeito fazem a interface com os dispositivos de suporte e comunicação definidos no documento.

keywords

Embedded systems, Integrated systems, mass storage, communications, USB, FPGA, Xilinx, Spartan 3E

abstract

This dissertation is written in the scope of ECU2010 project, and aims to determine the best possible solution for information storage and high speed communications for automotive motorsports applications.
The ECU2010 is centred on the research of a new architecture of electronic control units (ECU) for motor sport, focussing on control of internal combustion engines.
The proposed new architecture should be capable of controlling an internal combustion engine using the state-of-the art control models, but based on a distributed processing model consisting on self-sufficient processing modules in terms of communications, storage and intelligent enabled sensors/actuators, which of which is able to produce low-level data processing.
Communication between modules is not discussed herein, neither communication with the peripheral sensors/actuators. Instead, focus will be given to the communication between the processing modules and a control and monitoring device, hereinafter called the Host, that will be typically a personal computer or PDA.
This document will analyse and propose a solution for information mass storage and retrieval to a host system, mainly focused on historical data produced by variable monitoring and processing. The purpose of this document outcome is to produce a set of reconfigurable digital electronic IP cores, implementing features mentioned above in a Spartan 3E Xilinx FPGA.

Table of contents

Table of figures

1 Introduction

1.1 Summary

This chapter begins by defining the current thesis guidelines, followed by motivation and an overview of the objectives. It's completed by a brief description of the thesis organization.

1.2 Guidelines

Current embedded systems have been growing in size and complexity for the past few years, fuelled by the growth of microprocessors/microcontroller computing power, memory and the integration of more complex peripherals. Along with the reduction of production costs and power demands, the SoC is now a reality.

Along with the increase of transistor count in silicon devices, the size and complexity of high-density reconfigurable silicon devices, such as FPGA, have been increasing, allowing a feature rich environment to develop new embedded systems concepts.

The ground is set to promote the development of complex systems using the benefit of fast and reconfigurable digital devices which, after the system is set up and running and depending on the target application, could be beneficial to develop an ASIC based on the reconfigurable logic RTL. If not, the system could benefit from the reconfigurable nature of the devices allowing, for example, future upgrades/revisions.

1.3 Motivation

The need for a seamless ECU that can deliver capabilities such as plug-and-play, seamless usage and scalability is paramount for any prototyping environment.

One of such demanding environments is Motorsports, as racing teams often require fast software development times over cost effectiveness. In general this requires that the hardware platforms to be very flexible and, generally, over dimensioned, leading to hardware specialization, which in turn requires specialized teams for software development in a given platform.

With the multitude of motorsports applications, this specialization often leads to resource multiplication spanned across the multiple platforms, rendering the overall effectiveness low. This leads to situations where, for example, a single bit change in any motorsport ECU firmware generally costs a large hundred Euros, if not thousands, mostly in working hours

With the previous guidelines in mind, the course is set for this document, which focuses in the development of high performance IP cores for a prototype ECU developed in a joint venture with *Aveiro University*, *Kulzer Consultores Técnicos* and *Bosch Motorsports*. The objective is to provide a system in which calibration algorithm and system architecture changes are seamless, user friendly and fast to implement.

1.4 Objectives

1.4.1 General guidelines

The objective was to provide a set of IP cores which allowed easy integration, along with the smallest footprint possible and highest performance available. They had to be also sufficient versatile to empower the complete system to cope with the design parameters. In order to achieve this, it would be preferable to have:

- Scalable architecture;
- Off-the-shelf components usage;
- Usage of open architectures as much as possible;
- Reduced core footprint;
- High performance;

The goal of the developed work is to have building blocks of two major components in an automotive ECU:

1. **Communications**, which will in turn yield to separate projects:
    a. **Wired**: High bandwidth, scalable architecture and user-friendly;
    b. **Wireless**: Lower bandwidth than it's wired counterpart but with increased scalability;
2. **Non-volatile storage**: Scalable architecture, fast readout speeds (for data download purposes), capable of retaining large amounts of operational data;

The proposed ECU architecture is based on a distributed processing scheme. Each unit will have its own wired and wireless communications core as well as storage capability.

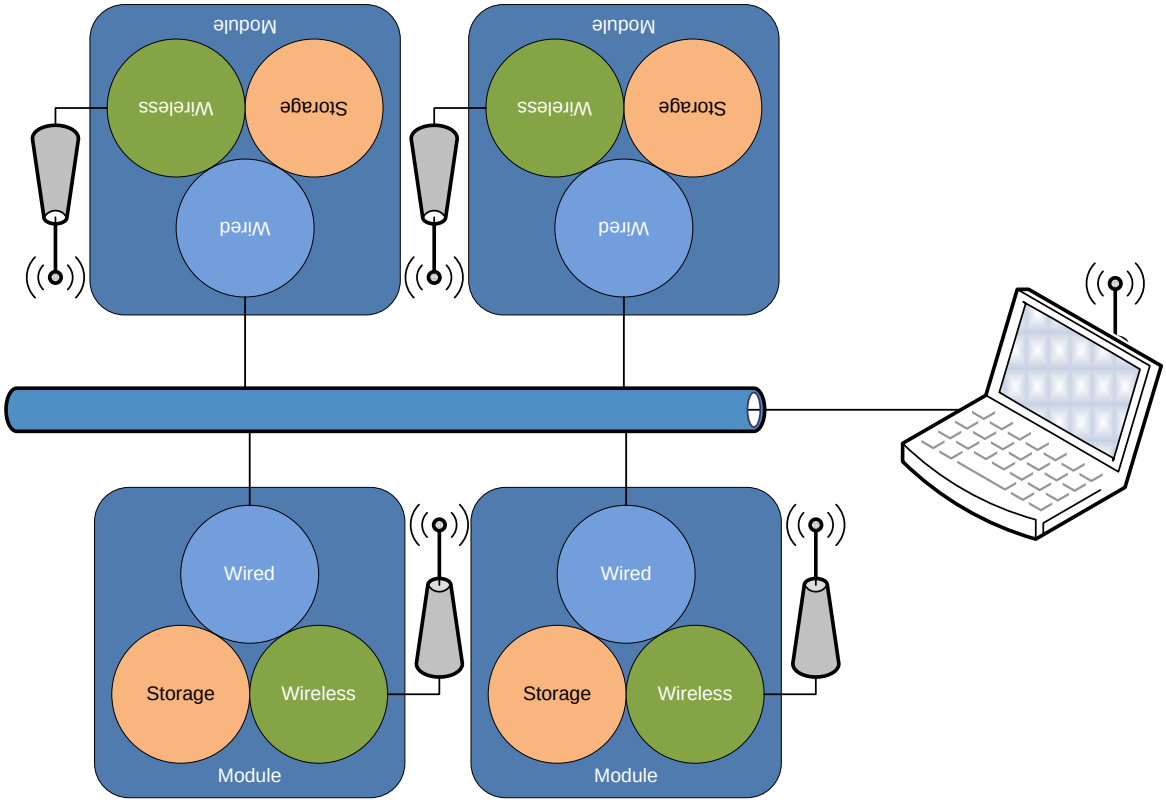


Figure 1 - Proposed ECU architecture

One of the advantages on the proposed distributed architecture is, for historical data recording purposes, to make the system scalable both in volume as in bandwidth, as the total volume available will be the sum of each module's volume capacity as with the bandwidth.

Up to some extent, the same is true for the wired and wireless communications, but in this area the overall system performance will greatly depend on the solution implemented. Some of the important aspects will be the transmission medium, sharing efficiency, maximum bandwidth and robustness to external interferences.

This is all true while there's the possibility to divide the overall functionality into several small blocks, which make the ECU control functions. The higher the subdivision, the higher the number of modules can process the same ECU control function collection.

1.4.2 Non-volatile storage

Non-volatile storage will be where the historical data, or datalogging, will be stored. It will be necessary to deploy a solution that is flexible enough to accommodate various combinations of number of variables to log and variable logging rate. Depending on the module capacity and logging rate, datalogging capacity can vary from the few hundreds of megabytes to a few gigabytes; nevertheless it would be beneficial to reduce datalogging download times as much as possible. Typical datalogging volumes would be on the thousands of variable range with a typical rate of one second for an accumulated time of a few hours, each variable will be a structure with timestamp, identifier and value totaling 10 bytes of length. A good assumption would be of about 35MB/ (hour × 1000 variables/sec).

1.4.3 Wired communications

Wired communications aim to be the primary communication mode with the ECU, being extensively used with ECU programming and control functions development. Being a distributed architecture, the global performance is the difference of accumulated module performance and inevitable interconnection losses. While trying to maintain interconnection losses to an absolute minimum, the whole system must be able to allow as much bandwidth as possible, because being a distributed architecture, the total amount of data to transfer could easily reach the dozens of Gigabytes. Optimal performance would allow downloading a typical datalog stream with one hour of records, ≈35MB, in less than 10 seconds; therefore it should have a minimum bandwidth of 7 MB/sec. This would assure that high volumes of data would be transferred within an acceptable time span.

1.4.4 Wireless communications

In contrast to wired communications, wireless communications do not aim to high performance, but to high scalability. This is so because of the need of low interoperation losses of multiple devices in close range, that could reach a few dozens in a single motorsport vehicle, once again due to the distributed nature of the system. Also, a low power solution would be preferable, as there is the need to have battery operated systems using the developed solution. Optimal specifications should have enough bandwidth to receive a datalogging stream of 35MB/ (hour × 1000 variables/sec) ≈ 80 Kbit/sec.

1.5 Organization

This document is organized in an introductory section, which defines de scope and constraints of the developed work.

It is then followed by an analysis of possible solutions to the presented problem, each one of them addressing a specific topic, and a comparison is made between them.

After the previous step, a functional description of each IP core developed is performed, as well a description of the solutions found to its implementation.

This document has a final section where the results are presented, discussed and from which conclusions are made.

All of the specific details about implementation, such as VHDL or C# code are present at the Annexes. Due to the sensitivity of the information, the Annexes are considered as confidential and may not be distributed with the rest of the documentation.

2 Analysis of possible solutions

2.1 Summary

In this section it will be described a set of different approaches available to each functional area. It is divided in two main focuses, one is the mass storage and the other is the external communications.

The analysis will try to evaluate the implications of each solution in the complexity of the associated IP core, given the designs constraints and application environment.

2.2 Mass storage

This functional area is of paramount importance for any motorsports ECU, providing precious data feedback to the Motronic engineering, so they can adapt and fine tune their algorithms.

The number of variables to monitor (either physical values from sensors, actuators or even intermediate calculation steps) grow exponentially with the ever increasing complexity of ECU algorithms, sensors and actuators.

Along with the growth of the number of variables to monitor, the sample rate also increases as the error threshold decrease and the demands for an ever increasing performance output from the Motronic systems increase.

The distributed architecture of this prototype ECU reduces the design constraints regarding the read/write bandwidth and capacity; nevertheless they should be as high as possible.

As such, the design constraints for mass storage IP core are:

- It must provide sufficient read/write bandwidth. Typical values will be on excess of 1MBps, for a centralized architecture. Exceeding this objective with some degree of scalability will leverage to the level of the most advanced ECUs available today.
- It must use a physical medium resilient to Motorsports environments; solid state support is strongly suggested, as the vibrations involved will typically destroy the fragile components of most mechanical storage devices.
- It must be as cost effective as possible: The usage of standard consumer mass storage devices is preferable. Such approach will increase the architecture survivability with mass storage technology evolution;

2.2.1 Flash cards

Since the introduction of the first commercial Flash chip by Intel Corporation in 1988 that we've seen the flash chips becoming a ever increasing common storage medium

Nowadays, and driven by the need of a high capacity, fast and cheap removable storage mediums, there're numerous of deployed formats as CompactFlash, SmartMedia, MultimediaCard, Secure Digital, Memory Stick and xD-Picture Card, with capacities ranging from 64MB to 32GB.

Any of the formats above described provide more than enough data retention and capacity, the following describes the pros and cons of each format for this application:

1. **CompactFlash**: Easy yet bulky parallel processor interface. Available cards are expensive and the edge that NOR technology gave in the early days of Flash media is now mitigated as the available Compact Flash cards in the market are in fact NAND devices with some interface to host microprocessor bus.
2. **SmartMedia**: Obsolete interface, hard to acquire new media.
3. **MultimediaCard**: Open standard reduces the cost of development of new IP cores as no royalty is required. High capacity new media available today, with interface bus bandwidth up to 52MBps (8 parallel data bits at 52 MHz);
4. **Secure Digital**: Closed standard greatly increases development costs, as royalties are enforced for non-SDA members. Not as fast as MultimediaCard, but with larger market penetration, making the acquisition of new media easier;
5. **Memory Stick**: Sony proprietary standard makes the development expensive, out of the scope of the project.
6. **xD-Picture Card**: just as Memory Stick, royalties make the development expensive, not supported by the scope of this project

As seen in the summary above, either CompactFlash or MultimediaCard could be used, mainly due to the tight budget of the project but, although CompactFlash is a well documented interface and still with a fair amount of devices available today, they're plain expensive medium.

In opposition to this, there's MultimediaCard, an open standard, with many available devices still in the market with no foreseeable obsolescence in the near future. The format in its latest revision supports high speed host bus interfaces, allowing a burst bandwidth of 52MBps, with standard (< 4GB) or high capacities (up to 32GB).

2.2.2 Conclusion

The MultimediaCard format offers all the needed characteristics, as the high capacity (for this application, anything over 1GB per processing cell is an overkill; 1GB would be enough to store over 268Msamples of 32 bit variables, this is over 3 days of data for a 1k samples/second application), the high speed attainable along with the availability of devices in the market and expectable lifespan of this format make it the obvious choice, fitting perfectly in the scope of this project.

Furthermore, the MMC protocol stack up to the application interface (MMC bus interface, read and write caches, macro command unit and power management) are relatively easy to implement, as will be discussed in section 3.1.

2.3 Communications

Communications with an ECU is a universal requirement, it would be impractical any kind of ECU that would not implement even the crudest communication technology.

Focusing on the objectives of this project, one of the main challenges of this chapter is to devise a communication scheme that, using available technologies, can reach proposed objectives.

Through careful design it is intended to provide the system with a communication system that is agnostic to the different implementations.

2.3.1 Wired communications

The main purpose of wired communications is to provide high bandwidth connectivity to the ECU. As such there's only a few number of technologies that could cope with this, as will be further analyzed.

2.3.1.1 IEEE 1394 Fire-wire

This is a multi master serial bus interface standard for high-speed communications and isochronous real-time data transfer and can connect up to 63 peripherals in a tree chain topology, supporting multiple hosts per bus. (1)

It allows peer-to-peer communications in order to reduce host load when performing peripheral to peripheral transfers, and designed to support plug-and-play as well as hot-swapping. (1)

FireWire devices implement the ISO/IEC 13213 specification model for device configuration and identification, commonly referred as the Configuration and Status Registers for microcomputer buses, to provide plug-and-play capability, including an IEEE EUI-64 unique identifier in addition device type and protocol support. (1)

Maximum cabling length is 4.5 meters with signal velocity at 5.05 ns per meter, and in the six circuit version, it supports any power configuration device, such as:

- Self-Powered Nodes
- Power Consumer
- Power Provider
- Alternate Power Provider

With the latter two configurations being capable to provide peripherals power up to 45 watts, with a maximum output voltage of 33 volts (1).

Release IEEE 1394-1995 specified FireWire 400 capable of data transfer rates up to 393.216 Mbps half-duplex. Later revisions of the specification include release IEEE 1394-

2002 which defines FireWire 800, 1600 and 3200, providing 786.432 Mbps, 1.6 Gbps and 3.2Gbps half-duplex transfer rates. (1)

The lack for manufacturer adoption, software support and implementation costs impaired this standard to penetrate a market other than high-end computing and professional applications. Consumer level implementations are scarce, and vendors usually prefer a cheaper and marginally less efficient solution – USB2.0

Further market penetration was impaired as Microsoft announced in December 4, 2004 that it would discontinue the support for IP networking over the FireWire interface in all future versions of Microsoft Windows (2). As a consequence, Microsoft's operating systems from Windows Vista onward lacked this support.

2.3.1.2 USB2.0

Universal Serial Bus aims to establish communication between peripheral devices a host controller which are agnostic to the peripheral device and transfer characteristics, while complying with bandwidth, latency and priority requirements. "The goal is to enable such devices from different vendors to interoperate in an open architecture" (3).

Early specifications of USB 1.0 defined a data rate of 12Mbps and 1.5MBps aimed for low bandwidth peripherals such as mice, printers and modems, as consumer electronics increased in performance and data storage capacity, USB 2.0 specification was released adding a new transfer rate of 480Mbps along with other major improvements.

Up to 127 devices can be attached to a single host controller in a tiered star topology, each tier having a hub, providing extra attachment points. All communications are initiated by the host, thus simplifying the device architecture as it does not need to know no other information about the remaining devices in the bus other than its assigned address. On the other hand, this strategy impairs the whole system as the host is required directly take part on all transfers, even if they're intended to be between devices, thus increasing its load between transfers.

This simplicity on the device implementation was of paramount importance to leverage USB over competitor's specifications, as FireWire. "USB has been the most successful interface in history of personal computing. Over 6 billion products are in the market, and over 2 billion ship a year now." (4)

Alongside with broad market adoption to the standard is a broad availability of microcontrollers, microprocessors and specialized interface integrated circuits, development tools and drop-in solutions. Extended operative system level support is also available, as there are standardized device classes that do not even require 3rd party drivers as major operative system already implements those functionalities.

2.3.1.3 IEEE 802.3 Ethernet

Ethernet is, as all other packet based communications, a frame-based technology, aiming to computer networking technologies for local area networks introduced by Xerox PARC at 1975 after developments started from 1973 by Robert Metcalf (5) inspired by the ALOHAnet.

In its first revision of 1980 provided 10Mbps bandwidth and 48 bit addressing capability and rapidly made its competitors, Token Ring and Token Bus, obsolete. By 1986 the largest computer network was ready with over 10000 nodes built around DEC's Unibus to Ethernet adapter (6).

The standard evolved from its first form 802.3 – 10BASE-5 providing 10MBps half duplex over an expensive RG-8/U coaxial cable to today's widely used 100BASE-TX and 1000BASE-T over inexpensive CAT5 twisted pair cable. In its latest form (June 2010), 802.3ba standard supports for example 100GBASE-ER4 which provides 100Gbps MAC speed over a 40Km single mode optical fiber with CWDM using four wavelengths channels around 1310nm of 25Gbps each.

This is the most widespread communication standard for computer networks, so availability of products as Ethernet modules, processor support, protocol stacks and operative system support is largely available.

2.3.2 Wired communications conclusions

FireWire was a good candidate, as it was able to deliver top notch capabilities to the system, but implementation costs, low availability of drop-in solutions and long development cycles impaired this solution to be adopted. Also, the level of peripheral complexity required greatly increases in order to take full advantage of the multi-master point-to-point communication scheme that FireWire empowers. Needless to say that every device needs to know the network topology and its attached devices in order to properly address their shared memory spaces and such approach would negate the simplicity of the module in favor of the whole system performance.

The Ethernet solution was very appealing at the beginning, and for simplification purposes a TCP/IP protocol stack was considered, as it would allow a seamless connection to most targeted Hosts, but that same stack demanded either the implementation of a reduced microprocessor running some proprietary firmware or even a reduced kernel such as Linux or the use of a embedded system bridging the gap between the ECU and the computer, both of them would be out of the scope of this project, as the latter would easily add a level of complexity that could easily become hard to manage as the system scalability started to grow. From all the solutions analyzed, this seemed to be the one that presented the most demanding technical challenges.

On the other hand, USB has shown to be sufficiently flexible and cost effective to broad deployment on consumer electronics, and is also capable of delivering high

performance connectivity. Its plug-and-play and hot-swap characteristics play an important role in the scope of this project, and on top of that, it's extremely easy to use from the end-user's point of view (7).

Given all of the above mentioned reasons, the path chosen to develop a wired communication solution is USB 2.0.

2.4 Wireless communications

This section acts as a complement to the communications module of our ECU, because wireless communications are not mandatory, they do have an advantage over wired communications regarding mobility and accessibility. Sometimes having a wire connecting to a laptop in a pit stop is just not an option.

2.4.1 IEEE 802.11 WLAN

This standard was first introduced at 1997 and supported two modes of operation, 1 Mbps and 2 Mbps, operating at the 2.4 GHz ISM band. Latest revisions to this standard include 802.11g that support transmission channels up to 54 MBps with forward error correction codes.

In general, 802.11g define 14 overlapping channels, of 22 MHz bandwidth:

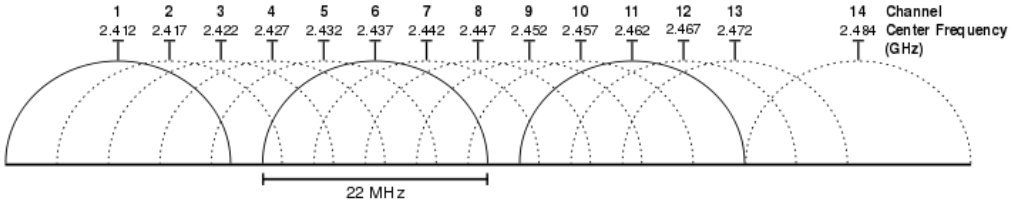


Figure 2 - IEEE 802.11g channels (8)

Channel usage is regulated by each country, although in Europe the adopted model uses channels 1 through 13.

Although de available bandwidth is high, the scalability is relatively limited by such adoption of overlapping channels. Theoretically only channels 1, 6 and 11 or 3, 8 and 13 can be used with minimum interference. Furthermore, 802.11g Clause 17 specifies a spectral mask that defines spectral power distribution, requiring the signal to be limited to -30 dB of its peak center power at ±11MHz from its center frequency, as represented on Figure 3.

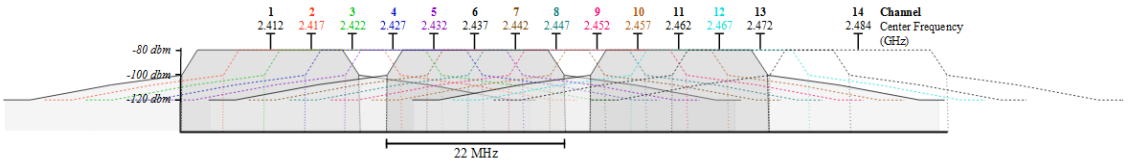


Figure 3 - 802.11g spectral mask (8)

Further enhancements were introduced in 802.11n amendment such as MIMO (multiple input multiple output) features, among others.

As it concerns to range, 802.11 is very dependant of the transmission power and transmission losses, but typical figures are in the 35m/140m for indoor/outdoor 802.11g to 70m/230m for 802.11n.

2.4.2 Bluetooth

Bluetooth uses the 2.4GHz ISM band, and in its simplest implementation, uses GFSK modulation with FHSS. This early versions had a Data rate of 1Mbps, although the most recent implementations allow up to 3MBps using 8DPSK modulation and has a range up to 100m under optimal conditions. It is based on the IEEE 802.15.1 standard. (9)

Bluetooth defines several profiles, each adapting the communication channel to a specific application, giving some degree of guarantees of available bandwidth and latency, thus allowing operating a multitude of devices under the same controller while minimizing the impact of data air time share.

Whereas 802.11 aims to replacing cabling in local area networks requiring higher setup complexity, Bluetooth aims to replace cabling of personal devices simplifying the setup procedure by using profiles. One of the simplest profiles is the SPP in which Bluetooth acts as a transport channel to a RS232 serial bus (9).

2.4.3 ZigBee

This protocol is based in the 802.15.4 and operates as 802.11 and Bluetooth in the ISM band, supporting low-power devices and simpler protocols than, for example, Bluetooth. (10)

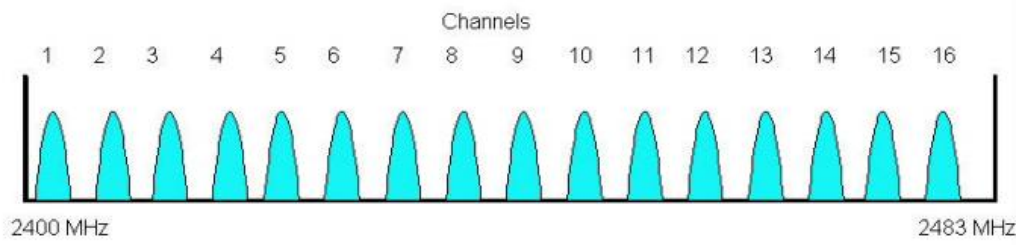


Figure 4 - ZigBee channel distribution (11)

Most implementations are available with integrated cost effective microcontrollers, allowing easy and rapid deployment in any design, being one of the most attractive features of ZigBee to have a small (in the order of a few milliseconds) wakeup time, thus reducing latencies for small transfers in contrast with, for example, Bluetooth.

With transfer rates up to 250kbps, ZigBee devices can easily deliver basic low-rate communications at a reduced price and power consumption.

2.4.4 Nordic Semi nRF24L01

This is a single chip transceiver, operating in the 2400 to 2483.5 MHz band with GFSK modulation and an air data rate up to 2Mbps. With integrated baseband protocol engine enabling packet communication, reducing the number of link-layer functions to be implemented by the application. Although the predefined output power range (-18dBm to 0dBm) is sufficient to most applications, it is possible to couple a RF amplifier to the chip in order to achieve transmission powers above 0dBm. (12)

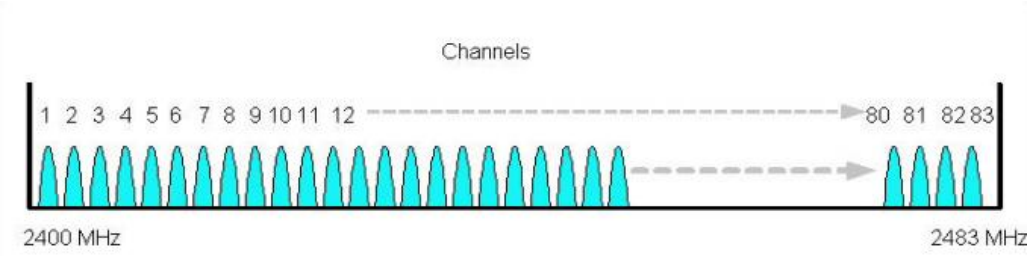


Figure 5 - nRF24L01 channel distribution (1Mbps mode) (11)

Being a highly integrated chip provides almost all of the necessary functions to operate. It has integrated voltage regulators that ensure a high PSRR allowing it to operate over a broad range of voltages. Also this device is able to operate from an inexpensive 60ppm 16MHz crystal which further reduces BOM costs. (12)

As referred above, this chip integrates baseband protocol engine enabling packet communications, referred as Enhanced ShockBurst™. It is a data packet link layer application, providing automatic packet assembly, timing, acknowledgement and retransmission of packets. It allows 1 to 32 byte dynamic packet size, and boasts 6 data pipe MultiCeiver™ for up to 1:6 node star networks. (8)

2.4.5 Wireless communications conclusions

The possibility to build peripherals with 802.11 standards is limited to the system capabilities, as the protocol stack is relatively complex and most implementations require the peripheral to run, at least, an embedded operating system.

Also the previous limitations of the allowable channels further impair the adoption of 802.11 as a wireless solution for this application, as it would be required that a few dozens of devices to operate in close proximity range. Such approach would increase

development cycles and costs and adding the scalability issues mentioned earlier makes 802.11 impractical to this application.

Bluetooth in the other hand aims to reduce peripheral complexity, but still require the host to be act as a master, increasing the host's complexity, thus impairing the implementations of inter-peripheral communications as needed in a wireless datalogger. Further impairments arise from the fact that in its basic form, Bluetooth master can only communicate with 7 slaves in a given network, limiting the number of devices that could be simultaneously connected to, for example, a personal computer, and therefore limiting system scalability.

Further improvements to Bluetooth impairments regarding latencies and setup complexity are made by ZigBee, sacrificing bandwidth and profiles. One interesting aspect of ZigBee is the availability of devices operating with transmission of 20dBm or more (10), which would allow communications between two devices to the a few kilometers range, being useful for telemetry for example.

The main issue in ZigBee is that the price range of such modules does not allow a solution to be cost effective, and the higher consumption/lower bandwidth efficiency of 802.15.4 DSSS make it less attractive than simpler protocol stacks. (11)

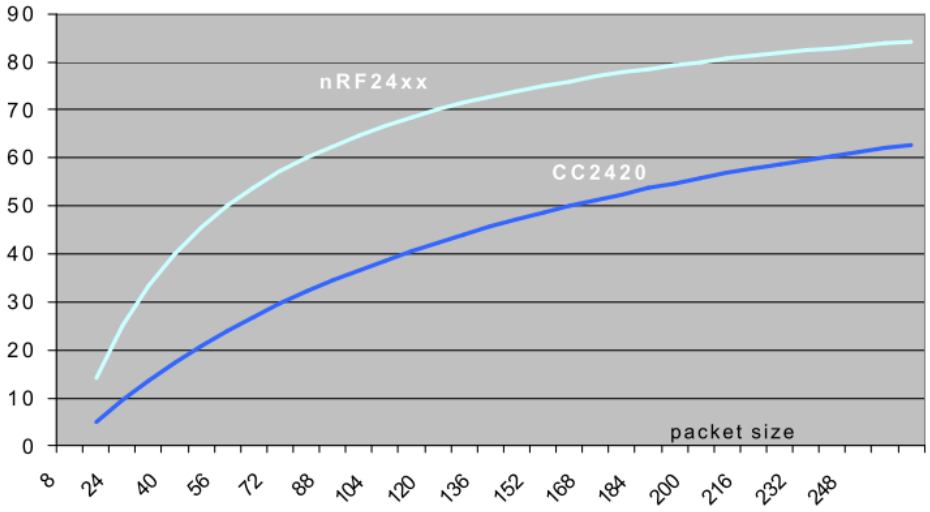


Figure 6 - nRF24L01 vs. ZigBee communication efficiency (11)

While being a proprietary stack, the Nordic solution, the nRF24L01 chip, is the solution of choice, as it delivers the needed functionalities while keeping software complexity to a minimum. Its low power characteristics, while not being paramount in this application, are desirable as to provide high mobility for a battery powered, datalog capable module.

The high scalability provides an advantage in this application, as it is possible to operate this device in 1MHz or 2MHz bands over the 2.4GHz ISM band, allowing up to 41

non overlapping channels to be deployed in close vicinity with a resolution of 1MHz. Even with overlapping channels, the Enhanced ShockBurst™ ensures packet transmission and retrial and, at the limit, takes full advantage of the fact that each device occupies roughly the channel at 60.6% capacity (maximum 164µs air time, with 130µs interval until acknowledgement is sent with 36,5µs air time minimum), thus easily accommodating two different communication channels. (12)

This device also has the capability to transmit data on the acknowledgment packets, although packet transmission is not guaranteed, which is very suitable for bidirectional signalization algorithms, and in addition to this, the nRF24L01 is able to rapidly change frequency and perform carrier sensing (about 300µs for a complete cycle), which allows to deploy frequency hopping solutions and/or or frequency spread spectrum capabilities. (12) (13) (14)

3 Description

Summary

This section describes the cores developed in this paper. It is divided by area of research and each area is described with a top-down approach.

3.1 Mass storage

3.1.1 MMC Electrical Standard, Standard Capacity (MMCA 4.1)

3.1.1.1 Introduction

"The MMC card is a universal low cost data storage and communication media" (15).

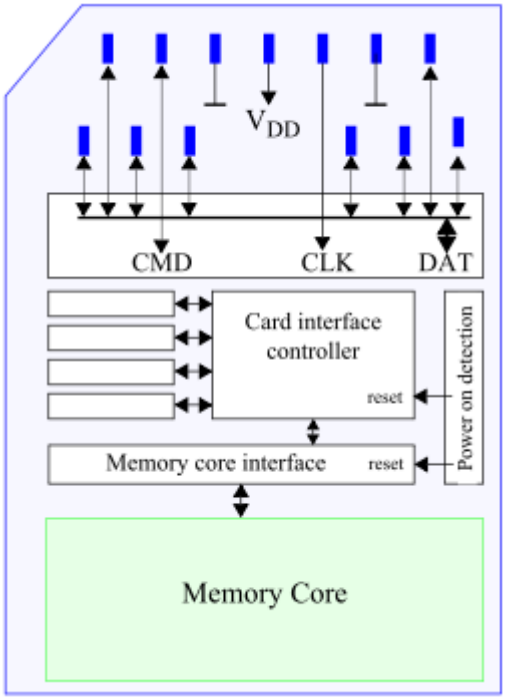


Figure 7 - MMC card architecture (15)

Based on a ten-wire bus topology, with speeds up to 52MHz yielding a maximum throughput of 416Mbps and is able to operate either at 1-bit, 4-bit or 8-bit bus width. It also supports lower bandwidth SPI bus (100-300 kbps).

The MMC bus lines are as follows:

CLK	Bus clock provided by the host
CMD	Bidirectional command line
DAT[0:7]	Bidirectional data line

Table 1 - MMC Bus lines

3.1.1.2 MMC Registers

The MMC implements 6 mandatory registers, as described below:

CID	Card IDentification number, a card individual number for identification
RCA	Relative Card Address, is the card system address, dynamically assigned by the host during initialization.
DSR	Driver Stage Register, to configure the card's output drivers.
CSD	Card Specific Data, information about the card operation conditions.
OCR	Operation Conditions Register. Used by a special broadcast command to identify the voltage type of the card.
EXT_CSD	Extended Card Specific Data. Contains information about the card capabilities and selected modes. Introduced in specification v4.0

Table 2 - MMC registers

3.1.1.3 MMC Transfers

Bus transfers are defined in the bus protocol as an exchange of messages between the host controller and the card, composed by tokens. These tokens are defined as:

Command	A command is a token which starts an operation. A command is sent from the host to a card. A command is transferred serially on the CMD line.
Response	A response is a token which is sent from the card to the host as an answer to a previously received command. A response is transferred serially on the CMD line.
Data	Data can be transferred from the card to the host or vice versa. Data is transferred via the data lines. The number of data lines used for the data transfer can be 1(DAT0), 4(DAT0-DAT3) or 8(DAT0- DAT7).

Table 3 - MMC tokens

The MMC standard supports two types of data transfers:

Sequential commands	These commands initiate a continuous data stream; they are terminated only when a stop command follows on the CMD line. This mode reduces the command overhead to an absolute minimum. Sequential commands

	are only supported in 1-bit bus mode.
Block-oriented commands	These commands send a data block followed by CRC bits. Both read and write operations allow either single or multiple block transmission. A multiple block transmission is terminated when a stop command follows on the CMD line similarly to the sequential read.

Table 4 - MMC supported transfers

The following section describes these transfers timing diagrams. For sequential reads it is as follows:

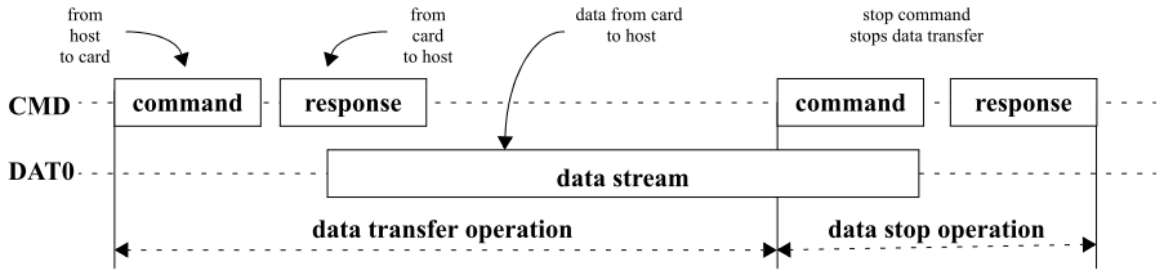


Figure 8 - Sequential reads (15)

The block reads have a similar timing diagram:

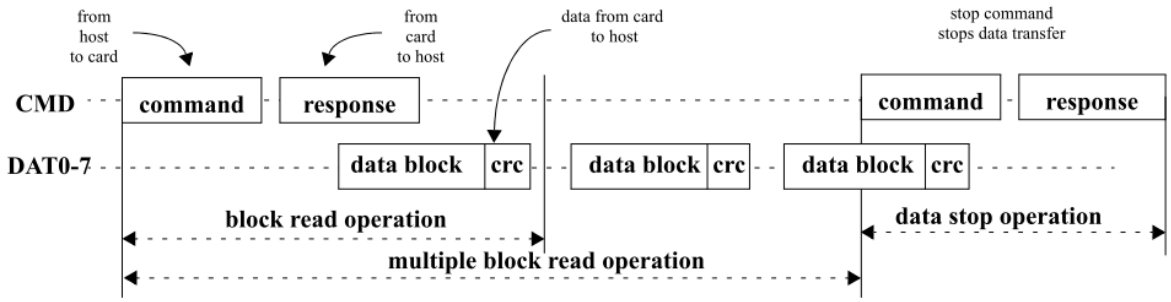


Figure 9 - Block reads (15)

As for writes, the sequential writes are as follows:

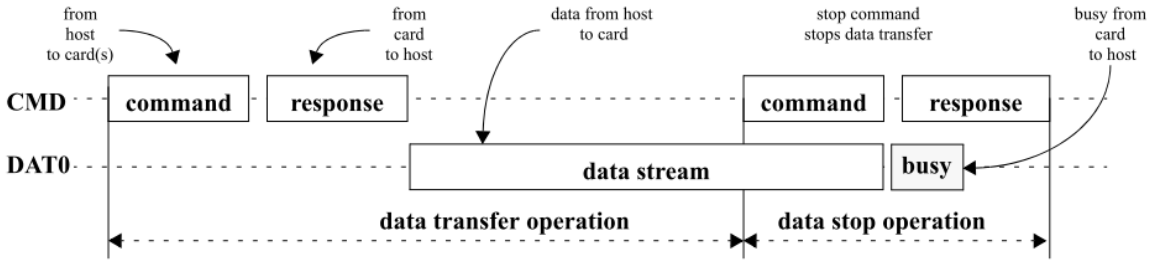


Figure 10 - Sequential write (15)

The block write operations are defined as:

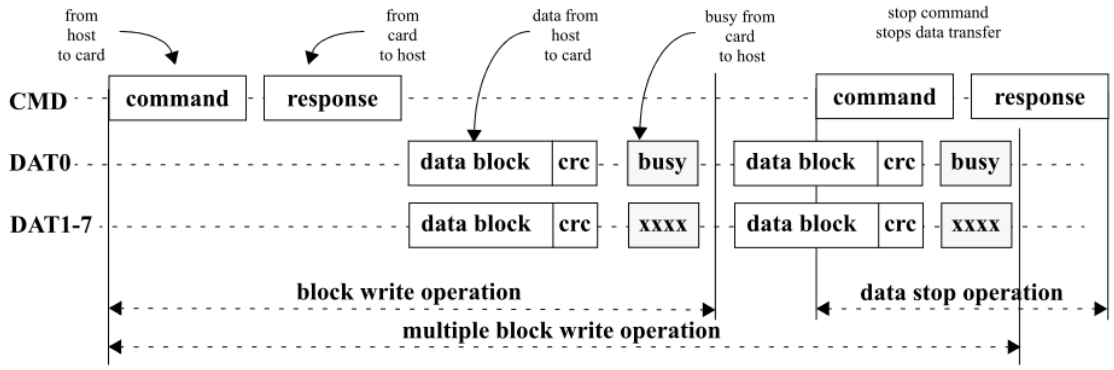


Figure 11 - Block write (15)

There are also some transfers that do not involve data transaction or even response, these transfers are as follows:

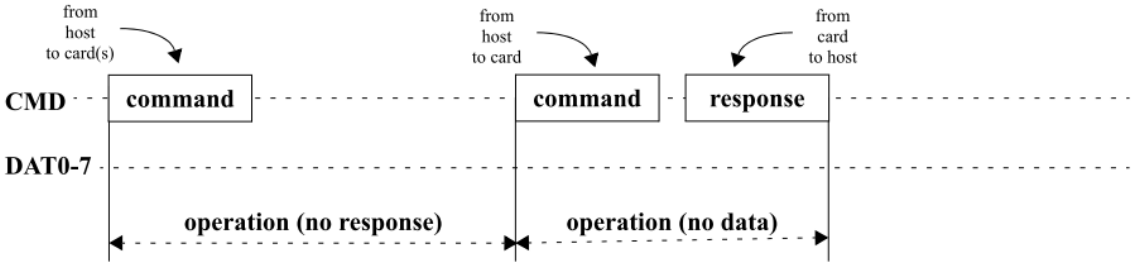


Figure 12 - No response and no data transfers (15)

3.1.1.4 MMC Token format

All tokens transfers are CRC protected, using either CCITT CRC7 (except for R3 responses or CID and CSD registers) or CRC16. All data transfers are protected using CRC16, with the exception of sequential data access.

The MMC token format is divided in two categories, command and response. The command format is defined as:

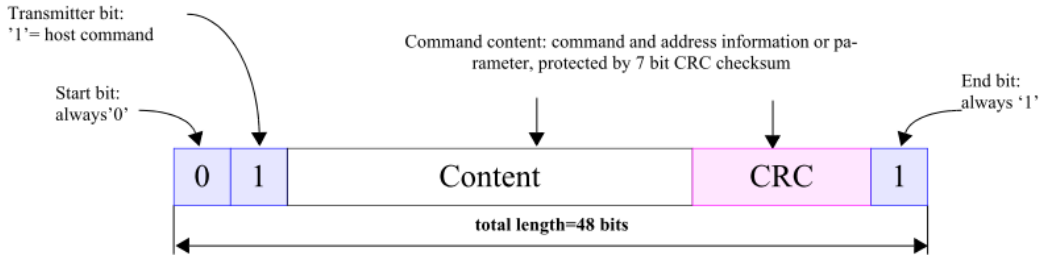


Figure 13 - Command token format (15)

The response token format is defined as:

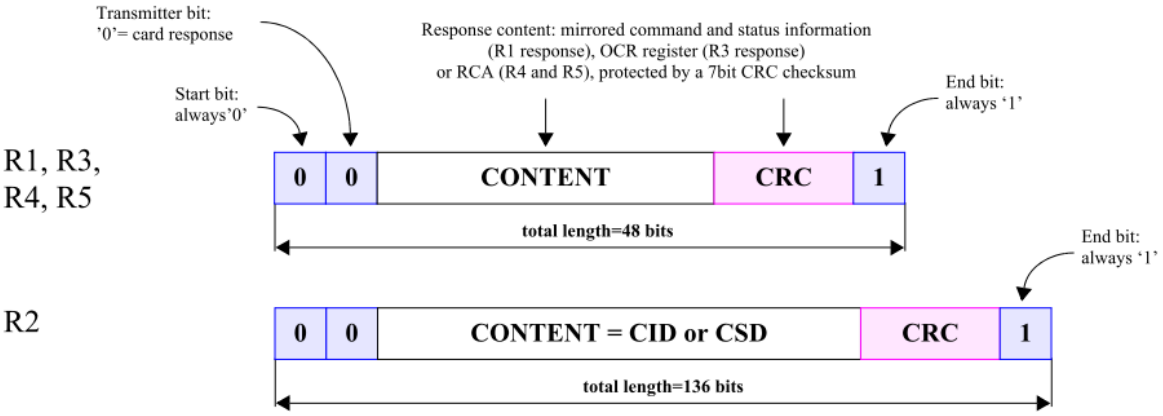


Figure 14 - Response token format (15)

All data transactions for 1-bit, 4-bit and 8-bit bus widths, regardless of the direction, are defined as follows:

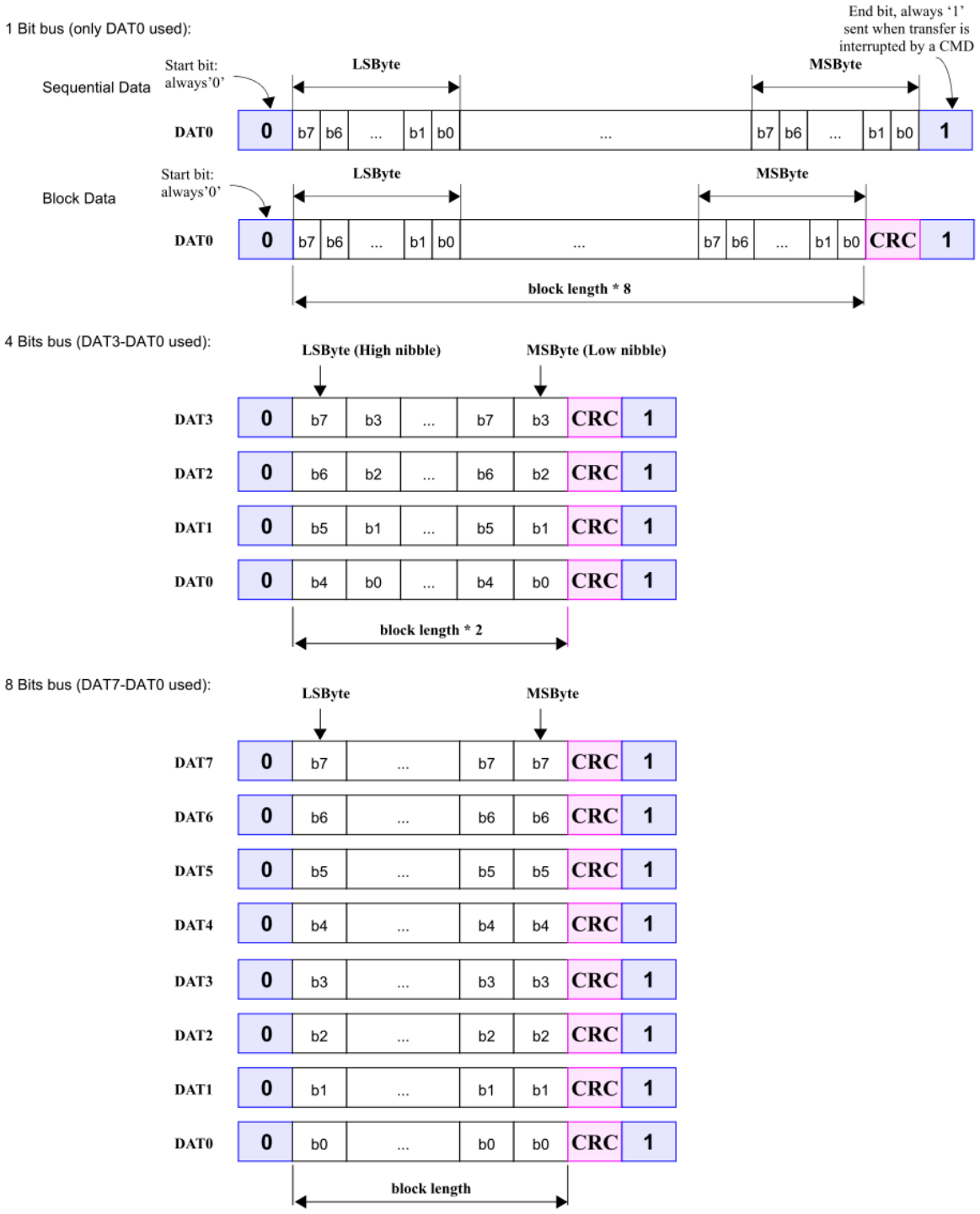


Figure 15 – 1-bit, 4-bit and 8-bit data packet format (15)

3.1.2 Mass storage overview

Mass storage was achieved implementing a MMC host on FPGA as follows:

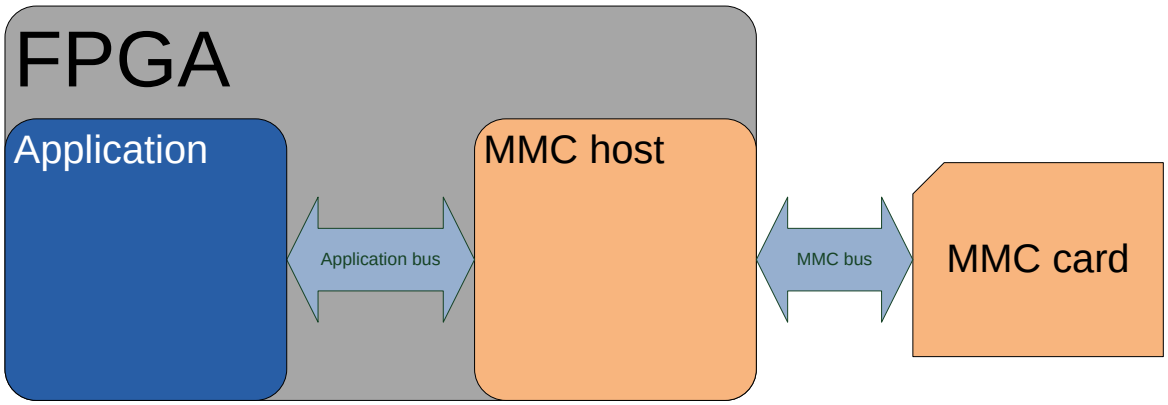


Figure 16 - Mass storage overview

The MMC bus offered several advantages over other solutions, such as:

- Availability of low cost MMC cards with more than enough performance (up to 52MBps burst bandwidth)
- Reduced number of lines needed to interface the FPGA to the MMC card
- Availability of documentation and application notes

The implemented MMC host has support for 1 bit MMC communications, up to 50MHz clock rate and single block transfers. The host structure overview is illustrated on

Figure 17.

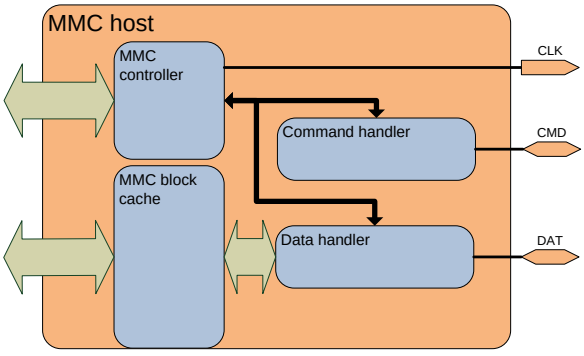


Figure 17 - MMC host overview

The MMC host has the following blocks:

- MMC block cache
- MMC controller
- Command handler
- Data handler

3.1.2.1 MMC block cache

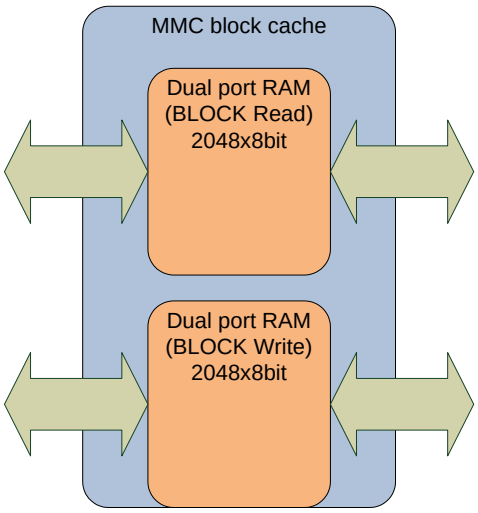


Figure 18 – MMC block cache overview

The MMC block cache has a total of 4kByte of RAM and is the ECU cell interface providing direct access to the MMC sectors.

In order to interleave read and write operations, as well as allowing having concurrent process to perform read and write operations on the MMC card, the MMC block cache has two independently addressable Dual Port RAM units. This leads for increased performance as the ECU cell using the MMC host is able to interleave read and write operations, transferring data from/to one half of the cache as the other is in use by the MMC host.

Each of the Dual Port RAM units has 2kByte address space (2048x8bit), so up to 2kByte block reads/writes are allowed.

3.1.2.2 MMC controller

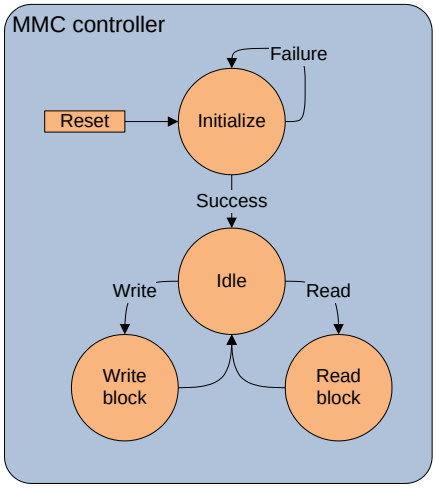


Figure 19 - MMC Controller overview

The MMC controller manages all aspects of the MMC bus such as Card Management, Card Interface Macros and MMC bus clock.

It also controls the remaining blocks of the MMC host during read or write operations.

After a system reset or power on reset, the card is initialized before usage, as the card state is unknown.

After successful initialization, the MMC controller idles until a read or write operation is requested by the ECU cell.

Any read or write operation is issued by the ECU cell to this block and the respective handshake signals are driven by this block, implementing basic handshaking to the ECU cell.

3.1.2.3 Command handler

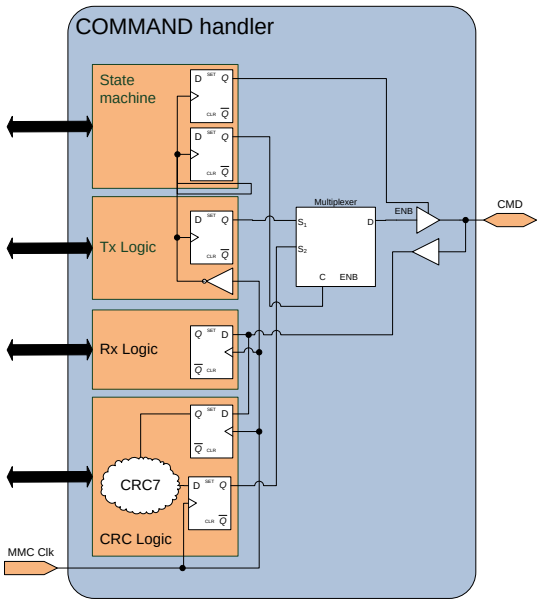


Figure 20 – Command handler overview

The command handler implements the bus level MMC protocol such as CRC calculation and generation, data inflation, data deflation and handshaking.

For this purpose, a small but fast finite state machine was implemented in hardware, in order to control all aspects referred above.

The advantage of this scheme is to encapsulate all of the bus level functions in one block to reduce complexity of the MMC host.

Note that the same CRC generator is used for both incoming/outgoing messages in order to reduce resource consumption.

3.1.2.4 Data handler

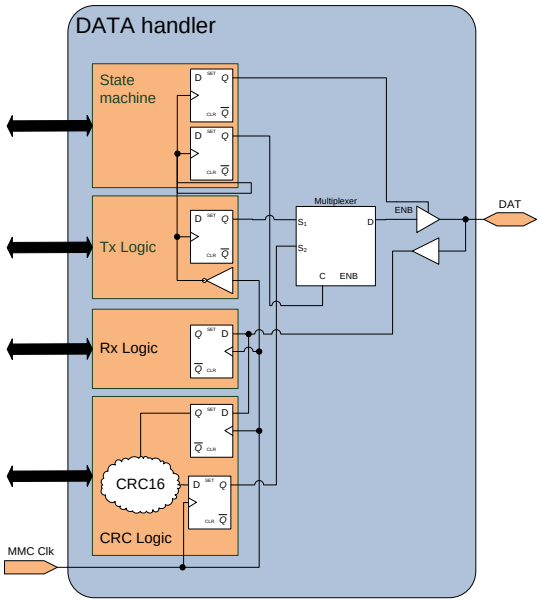


Figure 21 - Data handler overview

Data handler is similar to the command handler as the main difference is in the CRC block that is changed to match MMC specifications, and the data frames total length requires a more subtle approach for serial – parallel conversion.

Nevertheless, the similarity of the structures allows code reuse to some extent, reducing the overhead time to develop this block, maintaining the small footprint and fast responsiveness of the system.

3.1.3 Mass storage details

3.1.3.1 MMC block cache

3.1.3.1.1 Interface signals

The MMC block cache interfaces with the ECU cell with the following signals:

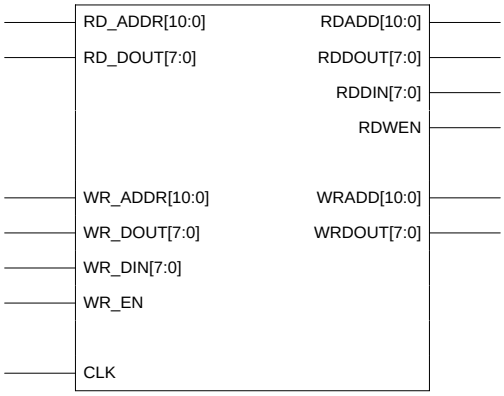


Figure 22 - MMC block cache interface

This block's interface signals summary is available on Table 5 - MMC block cache signals.

Signal Name	Width	Direction	Description
RD_ADDR	11	IN	Read block port a address
RD_DOUT	8	OUT	Read block port a data out
RDADD	11	IN	Read block port b address
RDDOUT	8	OUT	Read block port b data out
RDDIN	8	IN	Read block port b data in
RDWEN	1	IN	Read block port b write enable
WR_ADDR	11	IN	Write block port a address
WR_DOUT	8	OUT	Write block port a data out
WR_DIN	8	IN	Write block port a data in
WR_EN	1	IN	Write block port a write enable

WRADD	11	IN	Write block port b address
WRDOUT	8	OUT	Write block port b data out
CLK	1	IN	Clock in

Table 5 - MMC block cache signals

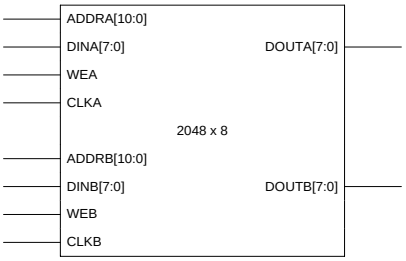
The block read operations have an 11 bit address bus (RD_ADDR[10:0]) and an 8 bit width data bus (RD_DOUT[7:0]).

To access the address space reserved for write operations it's available a 11 bit address bus (WR_ADDR[10:0]), a 8 bit data bus for write operations (WR_DIN[7:0]) and a 8 bit data bus for read operations (WR_DOUT[7:0]) along with a write enable signal (WR_EN).

The block read/write operation completion is signaled by the MMC controller, described further ahead, as such, during a block read/write operation it's not recommended to change the contents of the write block address space as written data will be unknown, as well the read block address space should only be read after the read operation completes successfully.

3.1.3.1.2 Building blocks – the Dual Port RAM

The implementation of the MMC block cache was based on the Dual Port RAM blocks available in the Spartan 3E FPGA. This allows greater throughput, and smaller footprint than a discrete solution.



The Spartan 3E Dual Port RAM was synthesized by using the Block Memory Generator at Xilinx® CORE Generator™ program, which is a design tool that delivers parameterized IP core optimized for Xilinx® FPGAs

Figure 23 – 2048 x 8 bit Dual Port RAM

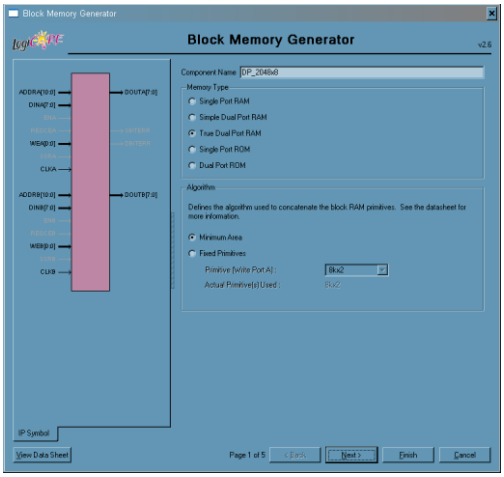


Figure 24 – Xilinx ISE Block Memory Generator

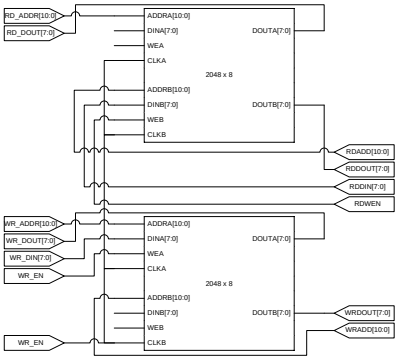


Figure 25 - MMC cache detail

As a dual-port RAM, the MMC block RAM allows both ports to simultaneously access the same memory cell. Potentially, conflicts arise under the following conditions:

1. If the clock inputs to the two ports are asynchronous, then conflicts occur if clock-to-clock setup time requirements are violated.
2. Both memory ports write different data to the same RAM location during a valid write cycle.
3. If a port uses WRITE_MODE=NO_CHANGE or WRITE_FIRST, a write to the port invalidates the read data output latches on the opposite port.

After analysis of the ECU cell requirements, it can be seen that:

- The clock domain is the same for each port, so condition 1 is not met.
- In this specific application, for each memory module there's only one process that modifies the contents of the memory cells, so the condition 2 is not met.
- As the read/write operations are signaled to the ECU cell by handshaking signals, read and write operations should be mutually exclusive in read or write block address areas. This is an assumption, as the ECU cell has to guarantee that there's no read / write collisions.

This leads to the conclusion that as long as the previous conditions are met, Dual Port RAM conflicts are not an issue in this design.

3.1.3.1.3 Signal sequence introduction

The implemented RAM is based on synchronous, positive edge triggered dual port RAM's, so read and write operations have the following sequence:

3.1.3.1.4 Read block address space signal sequence

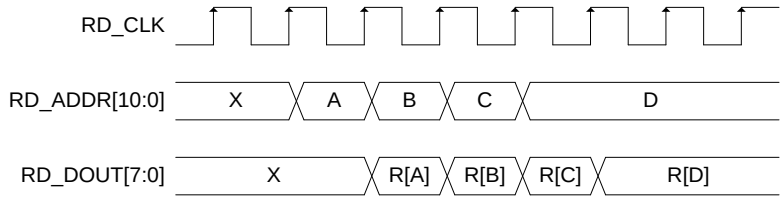


Figure 26 - MMC block cache read timing

As it can be seen, the RD_DOUT data changes after RD_ADDR is stable during a RD_CLK positive edge, thus introducing one cycle data latency.

This is the behavior of a standard synchronous RAM.

3.1.3.1.5 Write block address space signal sequence

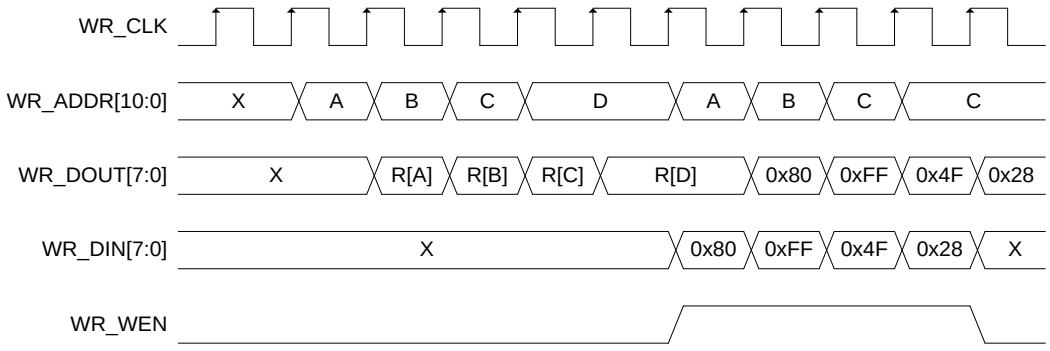


Figure 27 - MMC block cache write timing

As long as WR_WEN is logical ‘0’, the block RAM behaves has seen in the read sequence, being the WR_DOUT updated with the contents of the RAM with one cycle data latency. As the WR_WEN is driven to logical ‘1’, the addressed space in the RAM is written with the data present at WR_DIN bus at the next WR_CLK positive edge.

Also, this is the behavior of a standard synchronous RAM.

3.1.3.2 MMC controller

3.1.3.2.1 Interface signals

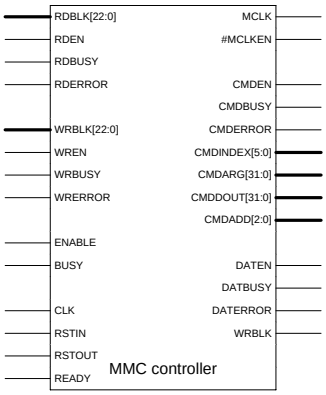


Figure 28 - MMC controller interface signals

The MMC controller provides the interface between the ECU cell and the command and data handler. Its signals are described on Table 6 - MMC controller interface signals.

Signal Name	Width	Direction	Description
RDBLK	23	IN	MMC block to read (23 MSB of 512 byte aligned address)
RDEN	1	IN	Drive logic ‘1’ for one clock cycle to start block read
RDBUSY	1	OUT	Logic ‘1’ while reading
RDERROR	1	OUT	Logic ‘1’ if read fails, reset at next block read
WRBLK	23	IN	MMC block to write (23 MSB of 512 byte aligned address)
WREN	1	IN	Drive logic ‘1’ for one clock cycle to start block write
WRBUSY	1	OUT	Logic ‘1’ while writing
WRERROR	1	OUT	Logic ‘1’ if write fails, reset at next block write
ENABLE	1	IN	Drive to logic ‘1’ to enable MMC controller

BUSY	1	OUT	Logic ‘1’ while not in Idle state
CLK	1	IN	Master clock input
RSTIN	1	IN	Reset input
RSTOUT	1	OUT	Reset output
READY	1	OUT	Logic ‘1’ after successful initialization
MCLK	1	OUT	MMC clock output
#MCLKEN	1	OUT	MMC clock output tri-state buffer control
CMDEN	1	OUT	Command handler enable signal
CMDBUSY	1	IN	Command handler busy signal
CMDERROR	1	IN	Command handler error signal
CMDINDEX	6	OUT	Command index
CMDARG	32	OUT	Command argument
CMDDOUT	32	IN	Command response
CMDADD	3	OUT	Internal register address
DATEN	1	OUT	Data serializer enable signal
DATBUSY	1	IN	Data serializer busy signal
DATERROR	1	IN	Data serializer error signal

Table 6 - MMC controller interface signals

3.1.3.2.2 MMC card Initialization

The MMC controller block implements the control finite state machine required for card initialization after power-up or reset and transfer management at each read/write cycle.

The MMC card initialization state diagram is defined as follows:

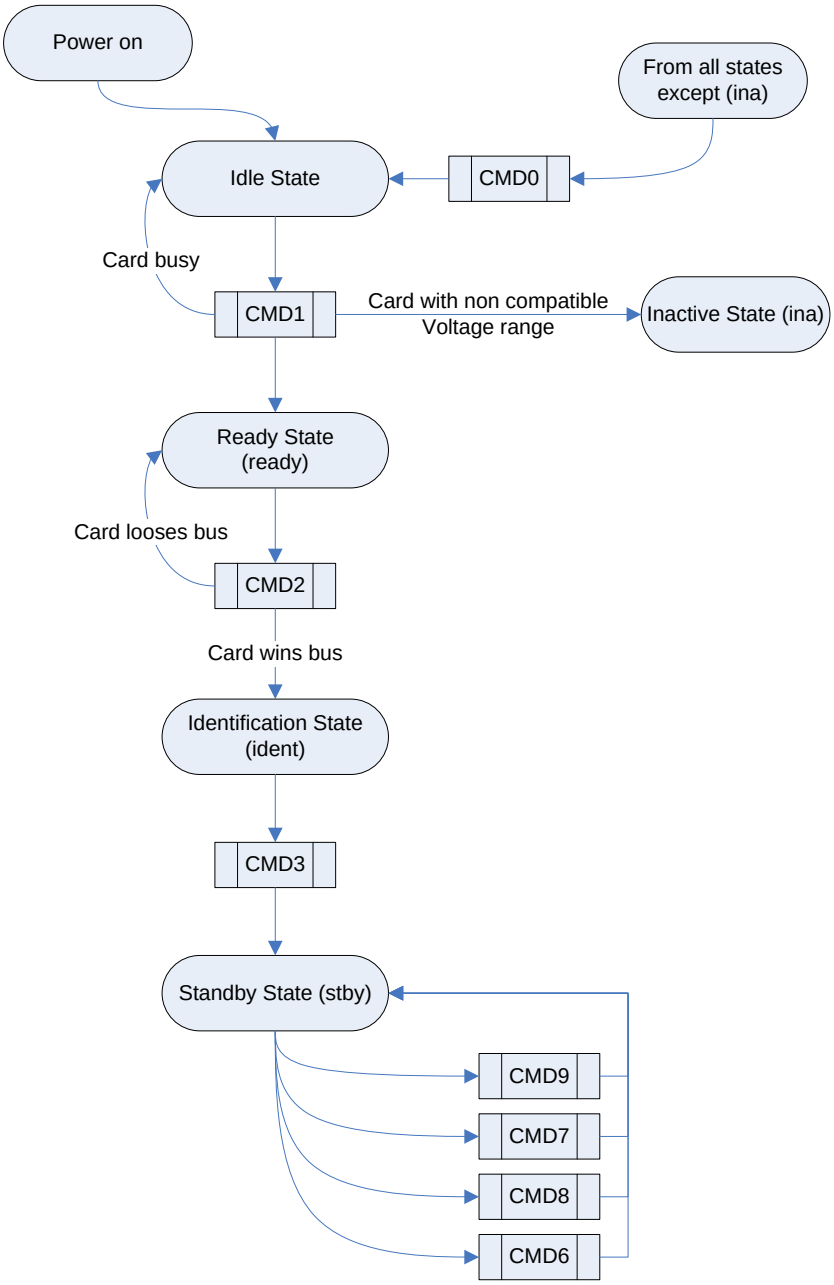


Figure 29 - MultimediaCard state diagram (card identification mode)

As the card state is unknown upon reset / power up, software reset (CMD0) is issued after a minimum of 1 millisecond of waiting time. After the software reset, the CMD1 is sent to the card, with the argument indicating that this is a high voltage card. Upon

successful completion of this command, the MMC card's OCR register is read using CMD2, followed by the card's CID (Card Identification Data) register readout using CMD3. As this is a single MMC card bus, the RCA (Relative Card Address) is assigned to 2 by default.

After successful initialization, the CSD (Card Specific Data) and the EXT_CSD (Extended Card Specific Data) are read from the MMC card and the card is switched to High Speed mode.

Synthesizing into a flowchart, the implemented initialization is as follows:

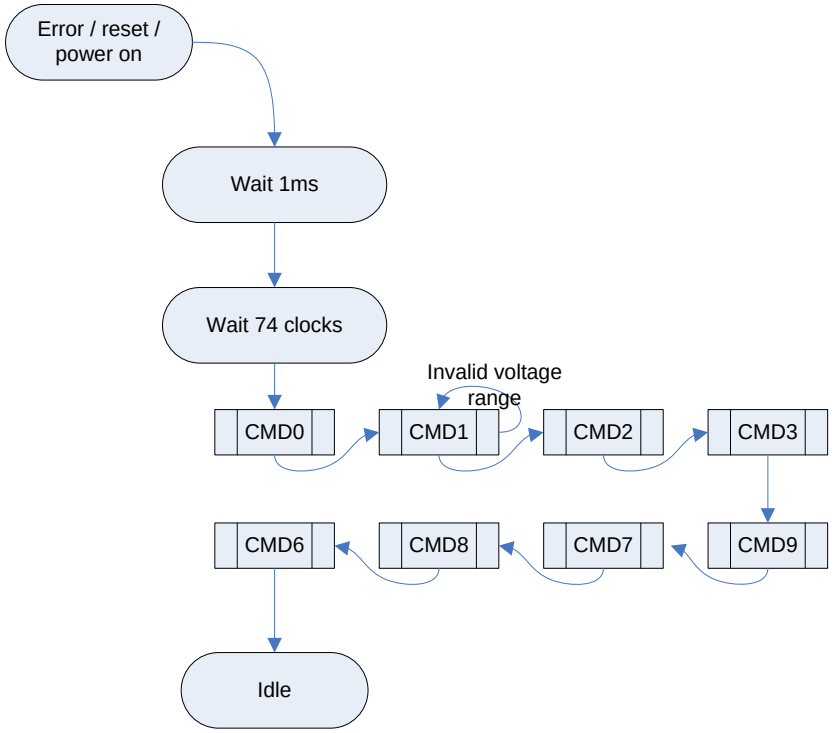


Figure 30 - MMC controller initialization

During initialization the CMD6 reads the EXT_CSD register from the MMC card. This register is not read as all the other registers (via CMD line), but as a data block (via DAT lines).

As such, and in order to reduce resource consumption and system complexity, the EXT_CSD register is read into the MMC Block Cache during the execution of CMD6.

This leads to if the ECU cell has the need to access the EXT_CSD data, the later is available right after initialization in the MMC Block cache lower 512 bytes address space (0x0000 – 0x01FF).

Note that the MMC block cache address space holding the EXT_CSD data is lost on the first block read operation, such data can be retrieved anytime by resetting the MMC controller, forcing the controller to re-initialize the MMC card.

3.1.3.2.3 MMC card basic operations - Block read

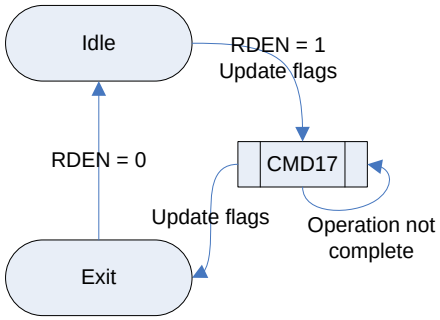


Figure 31 - Block Read

To start the block read sequence, the ECU cell drives the 23 MSBs of the target sector into the RDBLK bus, and asserts the RDEN signal, clearing the RDERROR bit and setting the RDBUSY bit.

Upon operation completion the RDBUSY flag is reset and the RDERROR flag is set accordingly to the operation result.

To initiate a second transfer the RDEN bit needs to be reset for at least one clock cycle.

During this operation the block read address space at the MMC block cache is being written as the data is shifted from the MMC bus, as such it is not recommended to read it until operation completion, otherwise data is likely to be corrupted.

The block write address space at the MMC block cache can be read at any time during this operation.

3.1.3.2.4 MMC card basic operations - Block write

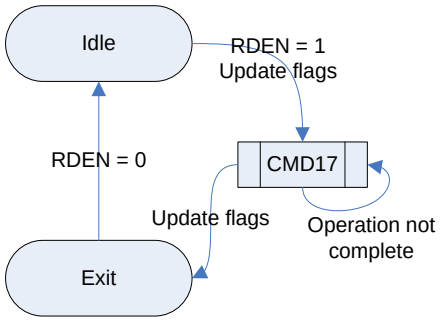


Figure 32 - Block Write

To start the block write sequence, the ECU cell drives the 23 MSBs of the target sector into the WRBLK bus, and asserts the WREN signal, clearing the WRERROR bit and setting the WRBUSY bit.

Upon operation completion the WRBUSY flag is de-asserted and the WRERROR flag is set accordingly to the operation result.

To initiate a second transfer the WREN bit needs to be reset for at least one clock cycle.

During this operation the block write address space at the MMC block cache is being read as the data is shifted to the MMC bus, and it is not recommended to modify it until operation completion, otherwise data corruption is most likely to occur at the MMC card.

The block read address space at the MMC block cache can be read at any time during this operation.

3.1.3.2.5 Error handling

The MMC controller block monitors the Command and Data handler error lines, and in any error event the current operation is aborted and the error is signaled to the ECU cell.

This allows for the ECU cell to decide whether to retry or to move on, based on the implemented algorithm.

3.1.3.3 Command handler

3.1.3.3.1 Interface signals

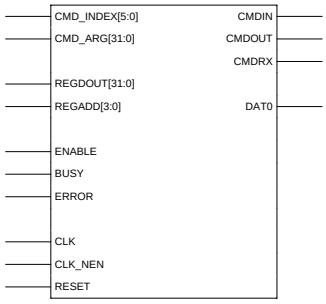


Figure 33 - Command handler interface signals

The signals available to control the Command handler are available on Table 7 - Command handler interface signals

Signal Name	Width	Direction	Description
CMDIN	IN	1	CMD line output – tie directly to IOB
CMDOUT	OUT	1	CMD line input– tie directly to IOB
CMDRX	OUT	1	CMD line output enable – tie directly to IOB
DAT0	IN	1	DAT0 line input – tie directly to IOB
CMD_INDEX	IN	6	Command index bus
CMD_ARG	IN	32	Command argument bus
REGDOUT	OUT	32	Response cache data out
REGADD	IN	4	Response cache address
ENABLE	IN	1	Command handler enable

BUSY	OUT	1	Command handler busy flag
ERROR	OUT	1	Command handler error flag
RESET	IN	1	Command handler synchronous reset
CLK_NEN	IN	1	Command handler clock enable
CLK	IN	1	Command handler clock

Table 7 - Command handler interface signals

3.1.3.3.2 Organization

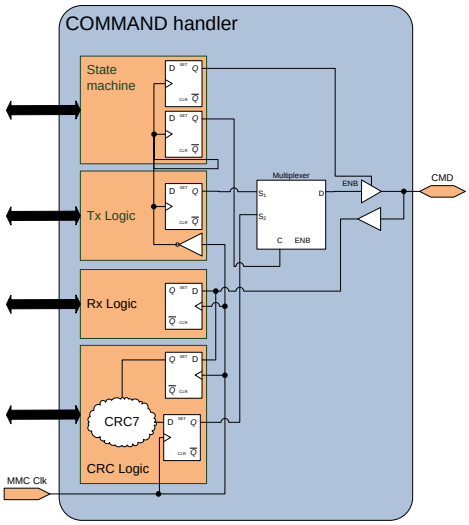


Figure 34 - Command handler overview

The command handler is built around two shift registers (one for incoming command frames and the other for outgoing command frames) and a common CRC generator with serial data output for both incoming and outgoing frame transfers.

A state machine binds all the components and provides for:

- Response type and length management
- Handshaking
- Error management
- Card register caching

3.1.3.3.2.1 State machine

This block has the task of building the transmission frame, switching the CMD output between the transmission frame shift register and the CRC generator output accordingly, detect the response length and handshaking for any given command index, verify incoming transmissions CRC.

3.1.3.3.2.2 TX Logic

This block is a simple shift register that holds the frame built by the state machine as the command was queued, and shifts-it out to the CMD line.

3.1.3.3.2.3 Rx Logic

As the TX Logic block, this block is a simple shift register and holds the complete received frame from the CMD line.

3.1.3.3.2.4 Response cache

This block is a 512 byte RAM used to cache the latest responses. This can be useful as the MMC host gets more complex and a broader range of MMC cards compatibility is desired. In this stage this cache can be used in order to configure the MMC parameters for optimal performance to a given MMC card.

The Response cache memory map is as follows:

Start Address	Register	Length (bits)
0x00	R1	32
0x01	R3	32
0x02	R2 (CMD2 or CMD10)	128
0x06	R2 (CMD9 only)	128
0x0A	R4	32
0x0B	R5	32

Table 8 - Response cache memory map

3.1.3.3.2.5 CRC Logic

The implemented CRC algorithm is standard CCITT-CRC7, as defined in the MMC Specification v4.1.

The CRC input is driven directly from the CMD line, so the same CRC generator can be used for both incoming and outgoing transmissions, thus reducing the logic consumption.

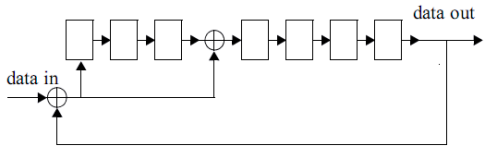


Figure 35 - CRC7 generator

In order to append the calculated CRC into the CMD line as the last part of the command frame being transmitted, the Command handler implements a multiplexer to the CMD line that takes the output from the Tx Logic as well the CRC generator output and is controlled by the state machine.

3.1.3.3.3 Operation

As the enable line is driven from logic ‘0’ to logic ‘1’, the CMD_INDEX and CMD_ARG status is stored into the transmission buffer, where the outgoing transmission frame is built. This enables the user to prepare the next command (CMD_INDEX + CMD_ARG) as the current is being transmitted.

In parallel, the CRC registers are cleared, in preparation to the pending transmission.

As the command frame bits are shifted out the CMD line, CRC is updated with the bit stream, enabling the state machine to shift out the newly calculated CRC at the end of the command frame.

At the end of the frame, the state machine verifies the response type to the currently shifted command, preparing the incoming reception handshaking and clearing the CRC registers.

Once the handshaking condition is met (if any), the reception shift register is activated by the state machine.

As the bits are shifted in, the state machine polls for the correct response length (which depends on the response type), once the response length condition is met, reception stops and received CRC is checked.

Received frame is valid if CRC is zero, so a simple wired OR is enough to test communication integrity.

On successful reception, the response is cached into the internal memory and the busy line is driven to logical ‘0’.

3.1.3.3.4 Error handling

The error sources are the following ones:

- Timeout
- CRC

CRC error sources are associated to the frame reception, as Timeout are generally related to CRC errors on the transmission, or bad command formatting. On the event of an error, the ERROR flag is driven to logical ‘1’ so that the MMC host is able to take containment measures, if possible.

3.1.3.4 Data handler

3.1.3.4.1 Interface signals

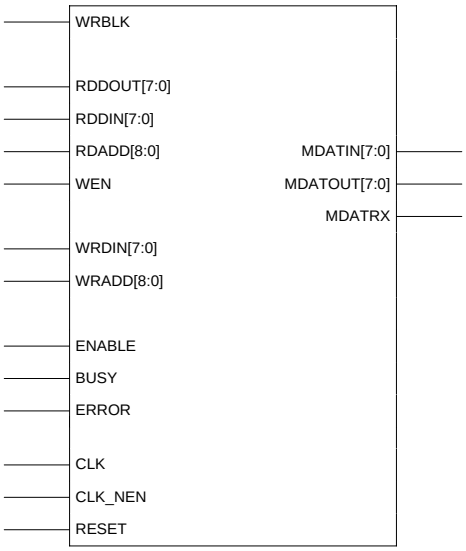


Figure 36 - Data handler interface signals

Signal Name	Width	Direction	Description
ENABLE	1	IN	Drive to logic '1' to enable data transfer
WRBLK	1	IN	Drive to logic '1' if a block write is to be performed
BUSY	1	OUT	Logic '1' during data transfer
ERROR	1	OUT	Logic '1' if last data transfer failed
RDDOUT	8	OUT	MMC block cache read block data out
RDDIN	8	IN	MMC block cache read block data in
RDADD	9	OUT	MMC block cache read block data address
WEN	1	OUT	MMC block cache read block data write enable
WRDIN	8	IN	MMC block cache write block data in
WRADD	9	OUT	MMC block cache write block data address

MDATIN	8	IN	MMC bus data in – tie directly to IOB
MDATOUT	8	OUT	MMC bus data out – tie directly to IOB
MDATRX	1	OUT	MMC bus data output enable – tie directly to IOB
CLK	1	IN	Data handler clock
CLK_NEN	1	IN	Data handler clock enable (active low)
RESET	1	IN	Data handler synchronous reset

Table 9 - Data handler interface signals

3.1.3.4.2 Differences to command handler

The basic concept is the same in these two blocks, two shift registers for each operation, frame reception and frame transmission, and one common CRC generator with serial data output.

As the MMC data bus has a slightly different handshaking method than the command bus, and the frame length is much larger, some changes are needed in this block

The main differences are:

- CRC generator. This block uses standard CCITT-CRC16, as defined in the MMC Specification v4.1 in opposition of the CCITT-CRC7 used for MMC commands.
- Data frame has 4096 payload bits plus start bit, CRC16 and stop bit; as such a shift register that holds the entire frame is expensive in resources. The solution is to use an 8 bit shift register and control the shift register load, start bit, crc and stop bits insertion.
- TX register parallel input is driven from the MMC block cache data output.
- Rx register parallel output drives the MMC block cache data input.

3.1.3.4.3 Organization

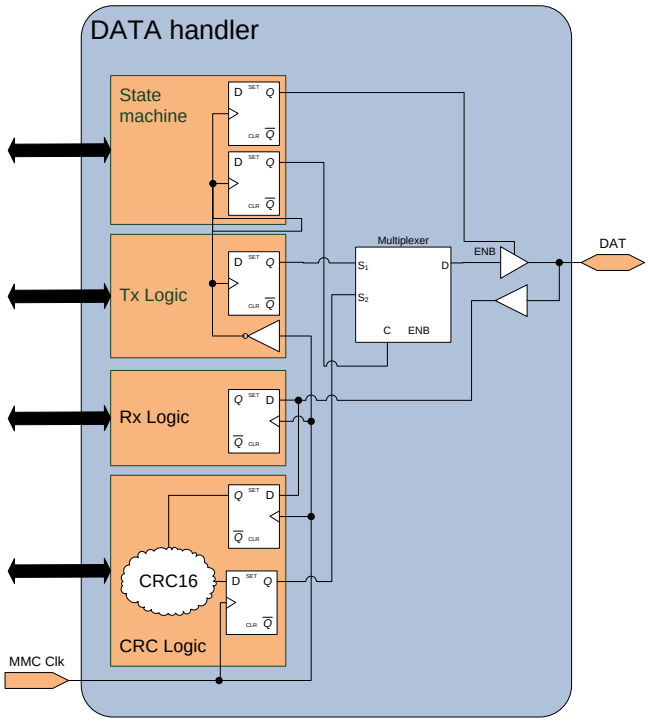


Figure 37 - Data handler overview

As the command handler, this block is built around two shift registers (incoming transmissions and outgoing transmissions) and a common CRC generator with serial data output for both incoming and outgoing frame transfers.

Also, a state machine binds all the components and provides for:

- Handshaking
- Error management
- Shit register – MMC block cache data path control

3.1.3.4.3.1 State machine

Similar to the command handler, this block builds the transmission frame, switching the CMD output between the transmission frame shift register and the CRC generator output accordingly, as well as inserting start and stop bits.

It also provides for handshaking and verifies incoming transmissions CRC, as providing data path control between the MMC block cache and the transmission / reception shift registers

3.1.3.4.3.2 TX Logic

This block is a simple 8 bit shift register, loaded with the current byte to transmit, of the standard 512 bytes frame length.

3.1.3.4.3.3 Rx Logic

As the previous block, this block is a simple 8 bit shift register, storing the received byte in preparation of storage in the MMC block cache.

3.1.3.4.3.4 Transmission / reception shift registers – MMC block cache datapath

One of the main differences between this block and the command handler is that the whole MMC data frame is not stored anywhere for both incoming and outgoing transmissions; the size of the frame (1 start bit + 4096 data bits + 16 CRC bits + 1 stop bit) makes that kind of solution impracticable.

As a solution to this problem, two 8 bit shift registers were implemented. The reason why 8 bit shift registers were selected is that this is the smallest data width supported by the Spartan's 3E Block RAM's used to implement the MMC block cache, while providing enough cycles between registers reload (8 MMC clock cycles).

The main difficulty with this kind of approach is that the data bits have to be shifted out without any latency cycles between them, and the block must be able to build a data frame as defined in the MMC specification 4.1.

3.1.3.4.3.5 CRC Logic

The implemented CRC algorithm is standard CCITT-CRC16, as defined in the MMC Specification v4.1.

Similar to the Command handler, the CRC input is driven directly from the DAT line, so the same CRC generator can be used for both incoming and outgoing transmissions, thus reducing the logic consumption.

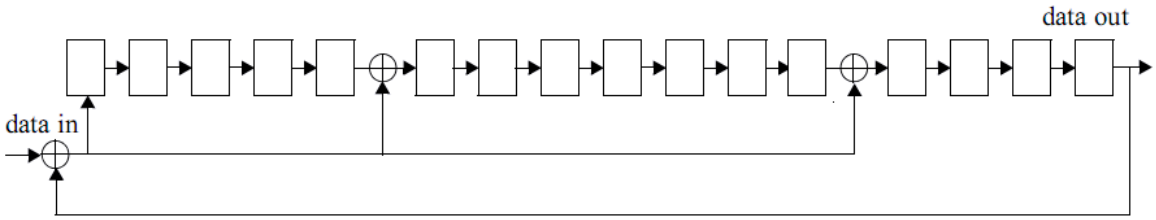


Figure 38 - CRC16 generator

In order to append the calculated CRC into the DAT line as the last part of the data frame being transmitted, the Data handler multiplexes the transmission shift register output with the CRC generator output and controls its output to the DAT line accordingly.

3.1.3.4.4 Operation

When the enable flag is driven from logical ‘0’ to logical ‘1’, the state machine verifies if the current operation is wither block read or block write.

In the event of a block read operation, the state machine waits for the frame start bit, after which it starts to shift in the data. Every 8 data bits, the shift register is stored in to the MMC block cache, until the 512 byte data frame reception is complete.

At the end of reception, CRC generator is verified, if the register is 0x0000, then the operation completes successfully, otherwise a communication error has occurred and the error flag is set to ‘1’, so the MMC host can take containment actions.

If the current operation is a block write, the state machine starts the operation by delaying the data frame by 2 cycles, after which it starts the data frame transmission by the start bit.

After the start bit, the state machine starts shifting the first byte out the MMC data bus, reloading the shift register every 8 bits. During this operation the CRC generator is calculating the CRC to be inserted at the end of the frame, after which the stop bit is sent.

Upon transmission completion, the state machine waits for the MMC card to send the CRC Token, validating the transmission. If the CRC Token is not received (timeout error) or the CRC token indicates that there’s a CRC error, the error flag is set to ‘1’, otherwise the state machine waits until the MMC card completes the write operation.

3.1.3.4.5 Error handling

The error sources are the following ones:

- Timeout;
- CRC;

CRC errors can occur either in transmission or reception, and is generally associated to MMC Bus line noise and transmission line characteristics.

On the event of an error, the ERROR flag is driven to logical ‘1’ so that the MMC host is able to take containment measures, if possible.

3.1.4 Mass storage operation

3.1.4.1 Summary

The implemented mass storage block is self contained, as it does not need any special configuration to initialize. It only needs a properly plugged MMC card.

In this section it is described the initialization procedure as well as this block operation.

3.1.4.2 Interface signals

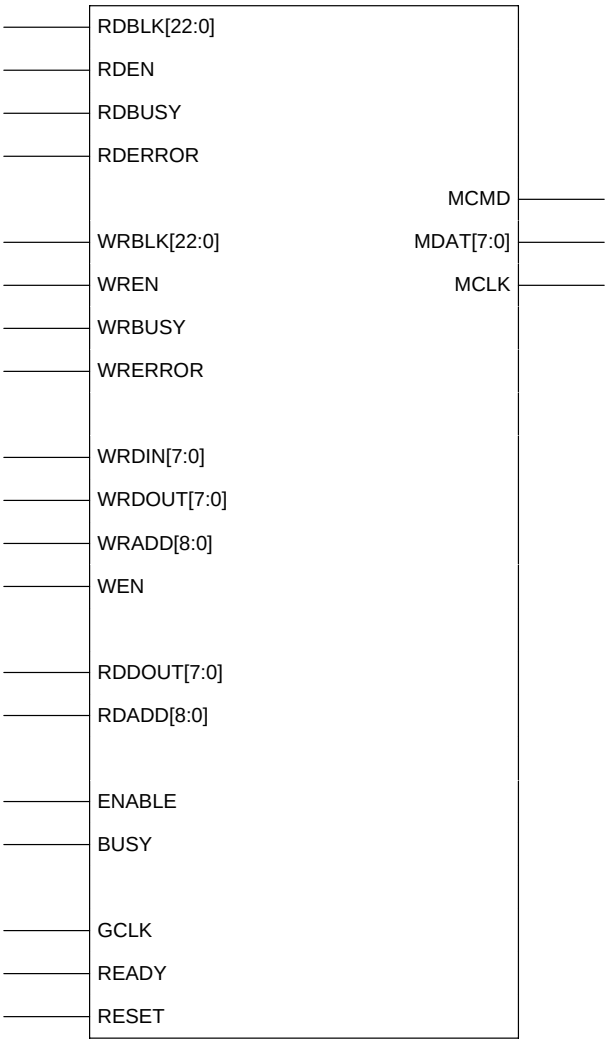


Figure 39 – MMC host interface signals

Signal Name	Width	Direction	Description
RDBLK	23	IN	Block to read – 23 MSB of 32 bit address
RDEN	1	IN	Drive to logical '1' to start block read
RDBUSY	1	OUT	Logical '1' while block read in progress
RDERROR	1	OUT	Logical '1' if last block read failed
WRBLK	23	IN	Block to write – 23 MSB of 32 bit address

WREN	1	IN	Drive to logical ‘1’ to start block write
WRBUSY	1	OUT	Logical ‘1’ while block write in progress
WRERROR	1	OUT	Logical ‘1’ if last block write failed
ENABLE	1	OUT	Drive to logical ‘1’ to enable MMC host operation
BUSY	1	OUT	Logical ‘1’ while not in idle
WRDIN	8	IN	MMC block cache data in – Block write address space
WRDOUT	8	OUT	MMC block cache data out – Block write address space
WRADD	9	IN	MMC block cache address – Block write address space
WEN	1	IN	MMC block cache write enable – Block write address space
RDDOUT	8	OUT	MMC block cache data out – Block read address space
RDADD	9	IN	MMC block cache address – Block read address space
MCLK	1	OUT	MMC bus clock
MCMD	1	OUT	MMC bus command line
MDAT	8	OUT	MMC bus data lines
READY	1	OUT	Logical ‘1’ after MMC card initialization
GCLK	1	IN	MMC clock used in high speed mode
RESET	1	OUT	MMC host synchronous reset

Table 10 - MMC host interface signals

3.1.4.3 Initialization

For MMC card initialization, the only action that needs to be taken is to drive the ENABLE signal to logical ‘1’ for at least one clock cycle. After that the MMC host independently initializes the MMC card and switches to high speed mode.

Successful initialization can be monitored at the READY signal. As it goes to logical ‘1’, the card is initialized, and if the BUSY signal is logical ‘0’, the MMC host is idling and ready for read/write operations.

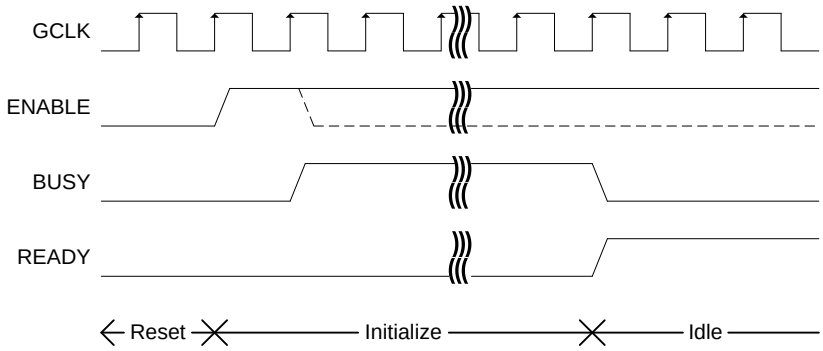


Figure 40 - MMC host initialization signal sequence

3.1.4.4 Block read

To start a block read, simply load the RDBLK bus with the upper 23 bits of the desired address to read, then drive the RDEN signal to logical '1'.

Completion of the operation can be done by monitoring the status of the RDBUSY signal. As it is logical '1', the operation is pending, if it's a logical '0' the operation has completed.

The success / failure of the last operation are taken from the status of the RDERROR signal. A logical '0' indicates that the read block operation was successful, otherwise an error has occurred. Note that to enable further block reads, the RDEN signal must be driven to logical '0' for at least one clock cycle after the operation completion.

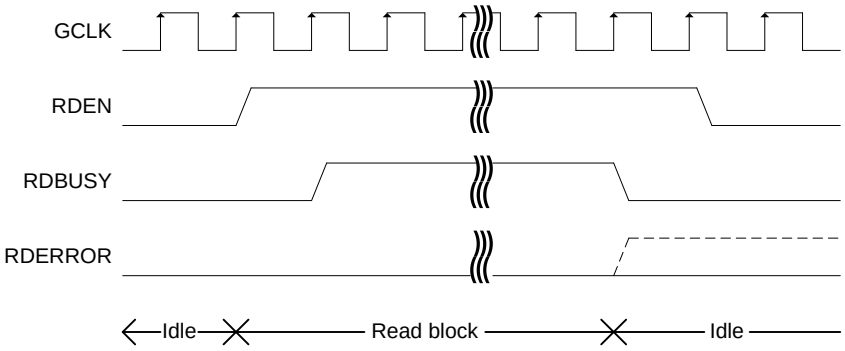


Figure 41 - Block read signal sequence

3.1.4.5 Block write

Similar to the block read, the block write is initiated by loading the desired block to be written into the WRBLK bus, followed by driving the WREN signal to logical '1'.

Operation completion given by the signal WRBUSY, as it's a logical '1' during the block write operation.

A logical ‘1’ at the WRERROR signal indicates that an error has occurred during the last block write attempt. Note that to enable further block writes, the WREN signal must be driven to logical ‘0’ for at least one clock cycle after the operation completion.

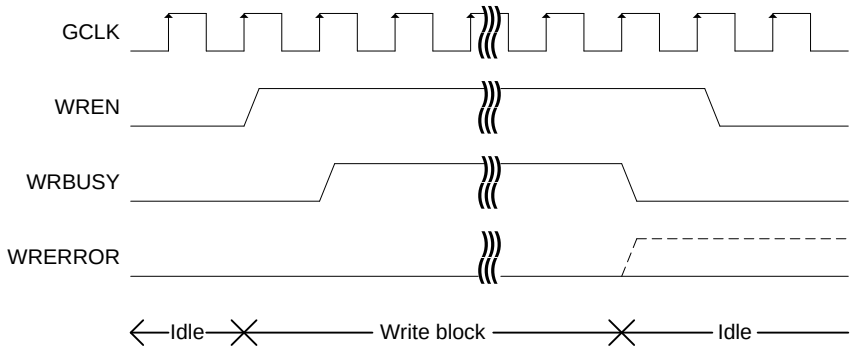


Figure 42 - Block write signal sequence

3.1.5 MMC host detailed schematic

The following figure illustrates how the MMC host blocks interconnect.

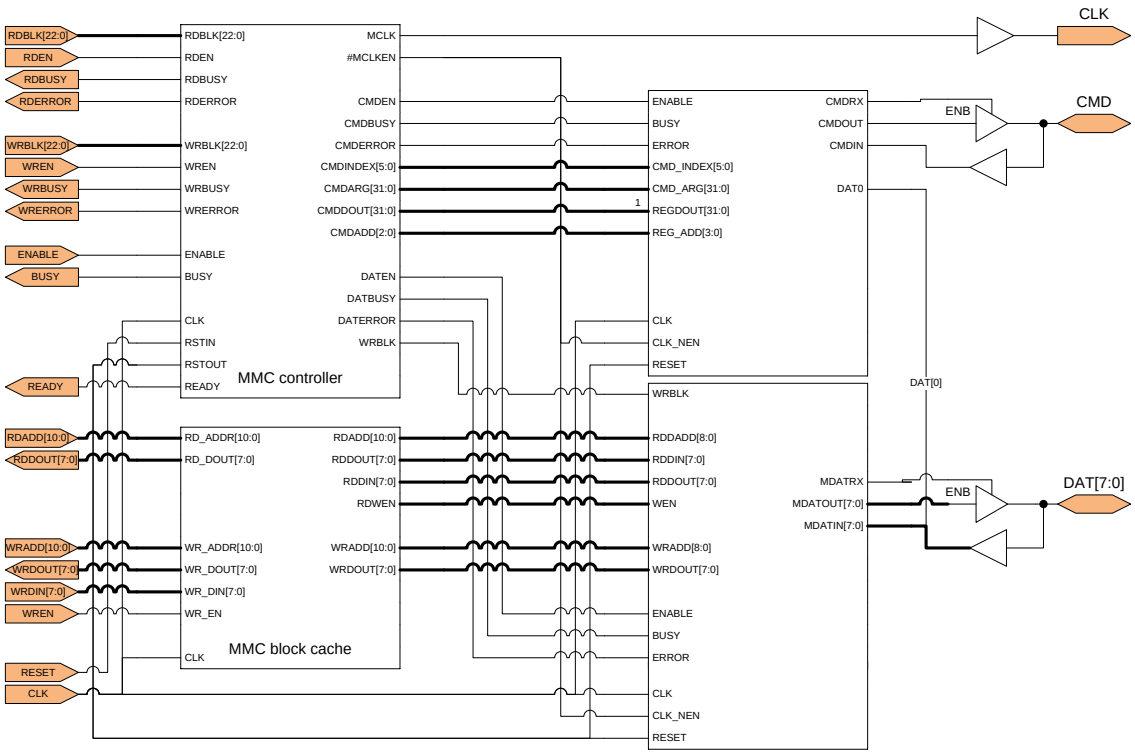


Figure 43 - MMC host schematic

At the left side of the schematic it is visible the ECU cell interface, on the right side it is visible the MMC card interface implemented through the Spartan 3E IOB’s.

3.2 Wired communications

3.2.1 Universal Serial Bus Specification Revision 2.0

"…USB continues to be the answer to connectivity for the PC architecture. It is a fast, bi-directional, isochronous, low-cost, dynamically attachable serial interface that is consistent with the requirements of the PC platform of today and tomorrow." (3)

3.2.1.1 USB Features

The USB Specification provides a selection of attributes that can achieve multiple price/performance integration points and can enable functions that allow differentiation at the system and component level.

1. Easy to use for end user:
    - Single model for cabling and connectors
    - Electrical details isolated from end user (e.g., bus terminations)
    - Self-identifying peripherals, automatic mapping of function to driver and configuration
    - Dynamically attachable and reconfigurable peripherals
2. Wide range of workloads and applications
    - Suitable for device bandwidths ranging from a few kb/s to several hundred Mb/s
    - Supports isochronous as well as asynchronous transfer types over the same set of wires
    - Supports concurrent operation of many devices (multiple connections)
    - Supports up to 127 physical devices
    - Supports transfer of multiple data and message streams between the host and devices
    - Allows compound devices (i.e., peripherals composed of many functions)
    - Lower protocol overhead, resulting in high bus utilization
3. Isochronous bandwidth
    - Guaranteed bandwidth and low latencies appropriate for telephony, audio, video, etc.
4. Flexibility

- Supports a wide range of packet sizes, which allows a range of device buffering options
- Allows a wide range of device data rates by accommodating packet buffer size and latencies
- Flow control for buffer handling is built into the protocol

5. Robustness
   - Error handling/fault recovery mechanism is built into the protocol
   - Dynamic insertion and removal of devices is identified in user-perceived real-time
   - Supports identification of faulty devices
6. Synergy with PC industry
   - Protocol is simple to implement and integrate
   - Consistent with the PC plug-and-play architecture
   - Leverages existing operating system interfaces
7. Low-cost implementation
   - Low-cost sub channel at 1.5 Mb/s
   - Optimized for integration in peripheral and host hardware
   - Suitable for development of low-cost peripherals
   - Low-cost cables and connectors
   - Uses commodity technologies
8. Upgrade path
   - Architecture upgradeable to support multiple USB Host Controllers in a system

3.2.1.2 Bus topology

USB physical interconnect is a tiered star topology; with a hub at the center of each star and point-to-point communications. Each hub can be connected either to another hub, compound device or function.

The standard limits the number of tiers to seven, in which the root hub is always connected in tier one. This limits the usage of compound devices to tier six, as compound devices are seen in bus topology as a hub with one or more functions attached.

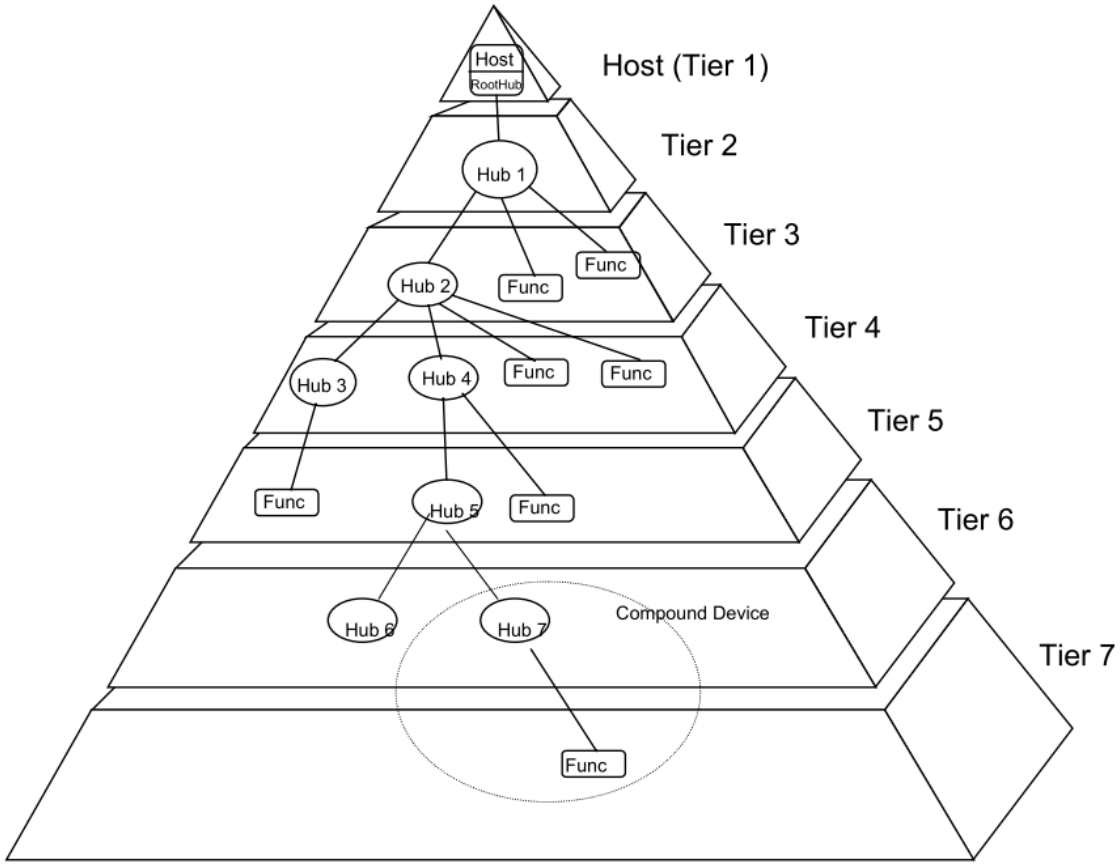


Figure 44 - USB bus topology (3)

USB Bus topology can be divided in four main components:

- Host and Devices - The primary components of a USB system
- Physical Topology - How USB elements are connected
- Logical Topology - The roles and responsibilities of the various USB elements and how the USB appears from the perspective of the host and a device
- Client Software-to-function Relationships - How client software and its related function interfaces on a USB device view each other

From a physical point of view the USB is a tiered star topology, in which Hubs play the role of interconnect, providing the bus attachment points

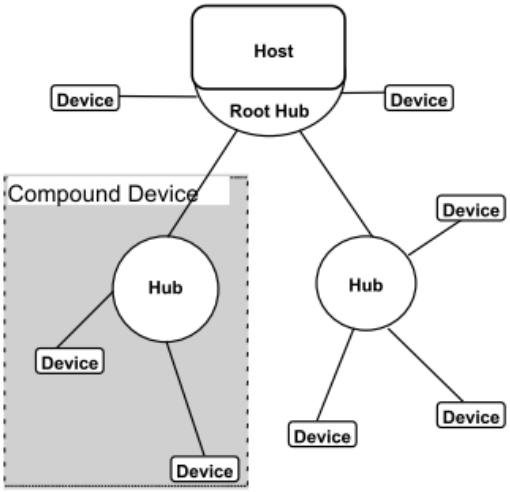


Figure 45 - USB Physical topology (3)

Here the hub plays an important role as to provide backward compatibility with previous implementations of the USB bus (Full and Low-Speed), yet providing the full benefits of the latest high-speed implementation. Therefore the Hub does not only allow interconnecting USB devices, as it also provides backward compatibility to existing devices, without large degradation of the USB system performance.

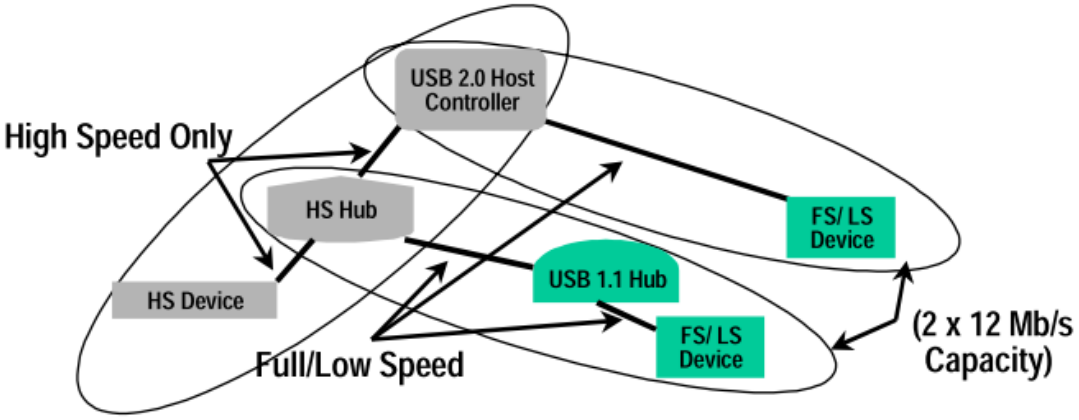


Figure 46 - Multiple speeds on the same system (3)

From a logical point of view, it can be said that, although the hubs are also logic devices and the host controller monitors the hubs for removal, the Host Controller communicates directly each logic device. This simplifies the communication model, as each logic device has a unique address assigned during device enumeration.

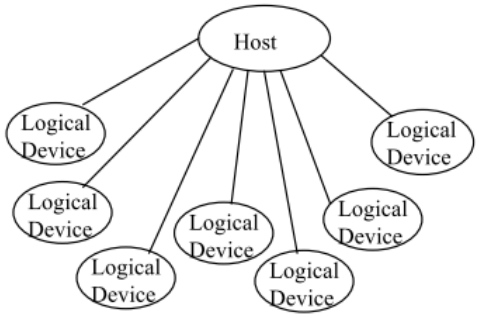


Figure 47 - USB logic bus topology (3)

In an even higher layer, the client software to function relationship can be seen as a peer-to-peer relation, one in which the Host is the master initiating all transfers and controlling all aspects of device enumeration and removal.

Even though the shared nature of the logic and physical bus topology, the client software manipulating a USB function interface is presented with the view that it deals only with its interface(s) of interest.

This simplifies the applications as they do not need to know the complete topology of the USB system and all of the attached devices.

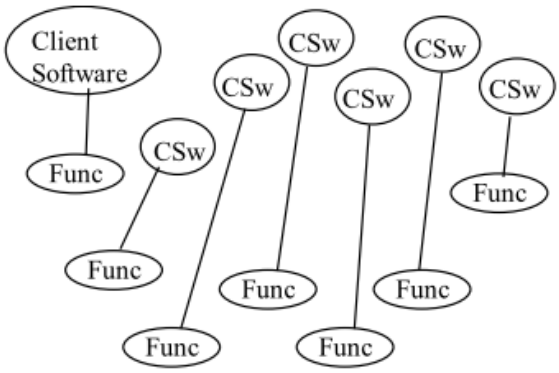


Figure 48 - Client software to function relationship (3)

Client software for USB functions must use USB software programming interfaces to manipulate their functions as opposed to directly manipulating their functions via memory or I/O accesses as with other buses (e.g., PCI, EISA and PCMCIA). During operation, client software should be independent of other devices that may be connected to the USB.

3.2.1.3 USB Host

For each USB system there's only one USB Host, which interfaces with the host computer system via Host Controller. Attached to the host system there's a root hub to provide one or more attachment points.

The host is responsible for the following:

- Detecting the attachment and removal of USB devices
- Managing control flow between the host and USB devices
- Managing data flow between the host and USB devices
- Collecting status and activity statistics
- Providing power to attached USB devices

The USB System Software on the host manages interactions between USB devices and host-based device software. There are five areas of interactions between the USB System Software and device software:

- Device enumeration and configuration
- Isochronous data transfers
- Asynchronous data transfers
- Power management

- Device and bus management information

The USB host occupies a unique position as the coordinating entity for the USB. In addition to its special physical position, the host has specific responsibilities with regard to the USB and its attached devices. The host controls all access to the USB. A USB device gains access to the bus only by being granted access by the host. The host is also responsible for monitoring the topology of the USB.

3.2.1.4 USB devices

There are two types of USB devices classes:

- Hubs – to provide extra attachment points to the USB; Hubs are wiring concentrators and enable the multiple attachment characteristics of the USB. Attachment points are referred to as ports. Each hub converts a single attachment point into multiple attachment points. The architecture supports concatenation of multiple hubs.
- Functions – to provide capabilities to the system; A function is a USB device that is able to transmit or receive data or control information over the bus. A function is typically implemented as a separate peripheral device with a cable that plugs into a port on a hub. However, a physical package may implement multiple functions and an embedded hub with a single USB cable. This is known as a compound device. A compound device appears to the host as a hub with one or more non-removable USB devices.

To assist the host in identifying and configuring USB devices, each device carries and reports configuration-related information. Some of the information reported is common among all logical devices. Other information is specific to the functionality provided by the device. The detailed format of this information varies, depending on the device class of the device, and is characterized by:

- Standard: This is information whose definition is common to all USB devices and includes items such as vendor identification, device class, and power management capability. Device, configuration, interface, and endpoint descriptions carry configuration-related information about the device.
- Class: The definition of this information varies, depending on the device class of the USB device.
- USB Vendor: The vendor of the USB device is free to put any information desired here. The format, however, is not determined by this specification

3.2.1.5 USB communication model

The simple view an end user sees of attaching one or more USB devices to a host is accomplished using the following model:

- The USB Bus Interface layer provides physical/signaling/packet connectivity between the host and a device.
- The USB Device layer is the view the USB System Software has for performing generic USB operations with a device.
- The Function layer provides additional capabilities to the host via an appropriate matched client software layer.

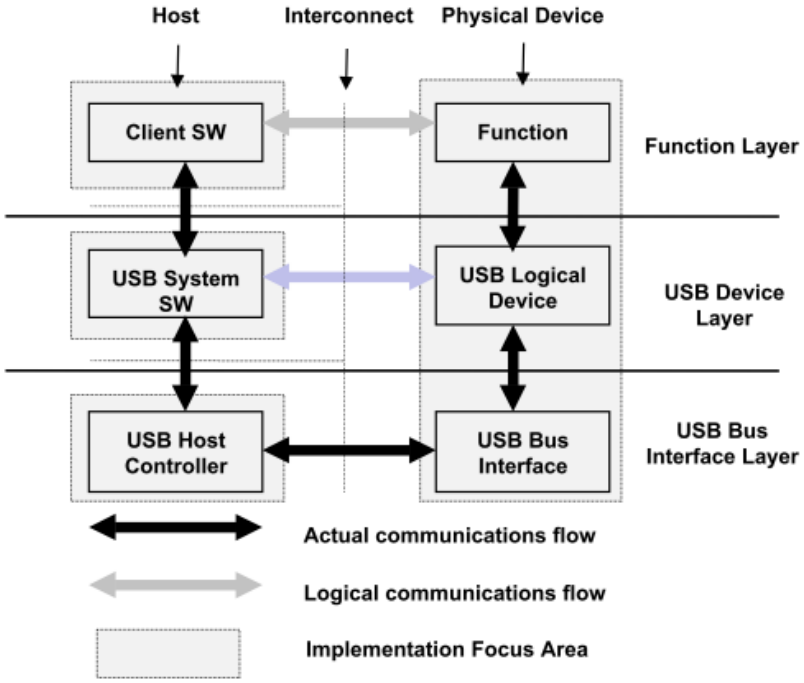


Figure 49 - Communication model (3)

The USB Device and Function layers each have a view of logical communication within their layer that actually uses the USB Bus Interface layer to accomplish data transfer.

The USB provides a communication service between software on the host and its USB function. Functions can have different communication flow requirements for different client-to-function interactions, enabling better overall USB bus utilization by allowing the separation of the different communication flows to a USB function.

Each communication flow makes use of some bus access to accomplish communication between client and function. Each communication flow is terminated at an endpoint on a device. Device endpoints are used to identify aspects of each communication flow.

3.2.1.6 Bus protocol

USB is a polled bus; the Host controller initiates all transfers, describing the type, direction, and device address and endpoint number of the transaction on a scheduled basis – this is referred as the token packet.

After this, the addressed USB device matching the address at the token packet selects itself.

The source of transaction (host or device) sends the requested data or signaling the host controller that no data is available.

After this the destination of the transaction responds indicating whether the transaction was successful – this is referred as the handshake.

USB data transfer model between the host and a device's endpoint is referred as a pipe, which are divided in stream pipes, with no associated data structure, and message pipes, with a defined data structure.

Each pipe is attached to a bandwidth, transfer, service type, and endpoint directionality and buffer size descriptor.

As the device is powered up, one message pipe (Default Control Pipe) always exists, providing access to the device's configuration, status and control.

The transaction scheduler allows the flow control for some data stream pipes. This is useful to prevent buffer overrun or underrun by deploying the use of NAK handshaking, thus enabling the flexible schedule of concurrent servicing, of a heterogeneous mix of stream pipes.

3.2.1.7 Error detection and handling

USB bus protocol uses separate CRC for control and data fields of each packet, capable to detect all one bit and two bit errors.

The usb controller retries a transmission with error up to three times, after which the client software is signaled and reacts to the error accordingly.

3.2.1.8 Device attachment, removal and enumeration

All devices are attached to the USB via a hub. These hubs report to the host controller the status of the ports. In the event of attachment, the host controller enables the hub port and configures the device through the default control pipe at its default address, after which the host controller assigns a unique address, notifying the client software in the case of a device, or, in the event of hub attachment, repeating the procedure to the devices attached to its ports.

With the removal of a device, the hub disables the port and when the host controller polls the hub's status bits so the USB System Software handles device removal.

Bus enumeration is the process by which the system identifies and assigns unique addresses to the attached devices, also detecting and processing device removal.

3.2.1.9 Data flow types

USB establishes a 1 millisecond time base called a frame on a full/low-speed bus and a 125 µs time base called a microframe on a high-speed bus. A (micro) frame can contain several transactions. Each transfer type defines what transactions are allowed within a (micro) frame for an endpoint. Isochronous and interrupt endpoints are given opportunities to the bus every N (micro) frames.

USB standard defines four types of data transfers:

- Control: Used to configure a device at attach time and can be used for other device-specific purposes, including control of other pipes on the device. It is possible to achieve up to 15.13 MBps bandwidth with 3% microframe usage per transfer.
- Bulk: Generated or consumed in relatively large and bursty quantities and have wide dynamic latitude in transmission constraints. It is possible to achieve a transfer rate up to 50.78 MBps with 8% microframe usage per transfer.
- Interrupt: Used for timely but reliable delivery of data, for example, characters or coordinates with human-perceptible echo or feedback response characteristics. It is possible to achieve a transfer rate up to 46.88 MBps with 42% microframe usage per transfer.
- Isochronous: Occupy a pre-negotiated amount of USB bandwidth with pre-negotiated delivery latency. It is possible to achieve up to 46.88 MBps bandwidth with 41% microframe usage per transfer.

A pipe supports only one of the types of transfers described above for any given device configuration, allocating a certain amount of bandwidth on establishment which, along with transfer type, defines bus access allowance.

3.2.1.10 Device endpoints

An endpoint is a uniquely identifiable portion of a USB device that is the terminus of a communication flow between the host and device.

Each logical device is composed by a collection of independent endpoints with a device-determined direction of data flow; each one is given at design time a unique device-determined identifier called the endpoint number.

The combination of the device address, endpoint number, and direction allows each endpoint to be uniquely referenced.

Each endpoint supports data flow in one direction: either input (from device to host) or output (from host to device).

An endpoint has characteristics that determine the type of transfer service required between the endpoint and the client software. An endpoint describes itself by:

- Bus access frequency/latency requirement
- Bandwidth requirement
- Endpoint number
- Error handling behavior requirements
- Maximum packet size that the endpoint is capable of sending or receiving
- The transfer type for the endpoint (refer to Section 5.4 for details)
- The direction in which data is transferred between the endpoint and the host

All USB devices are required to implement a default control method that uses both the input and output endpoints with endpoint number zero. The USB System Software uses this default control method to initialize and generically manipulate the logical device as the Default Control Pipe. The Default Control Pipe provides access to the device's configuration information and allows generic USB status and control access.

A USB device that is capable of operating at high-speed must have a minimum level of support for operating at full-speed. When the device is attached to a hub operating in full-speed, the device must:

- Be able to reset successfully at full-speed;
- Respond successfully to standard requests: set_address, set_configuration, get_descriptor for device and configuration descriptors, and return appropriate information;

The high-speed device may or may not be able to support its intended functionality when operating at full-speed.

Functions can have additional endpoints as required for their implementation. Low-speed functions are limited to two optional endpoints beyond the two required to implement the Default Control Pipe. Full-speed devices can have additional endpoints only limited by the protocol definition (i.e., a maximum of 15 additional input endpoints and 15 additional output endpoints). Endpoints other than those with endpoint number zero are in an unknown state before being configured and may not be accessed by the host before being configured.

3.2.2 Wired communications overview

Each ECU Cell is capable of processing a large number of operations, store a large amount of data, as it is also capable of near real-time debugging and prototyping. All of the capabilities require a transport layer able to cope with the bandwidth requirements, scalability as well as to enable ECU Cell hot swap. In addition, a well documented, industry proven and toughened bus with a broad variety of silicon devices ready to use is required, while reducing to a minimum the development time.

For all of these reasons, USB bus was selected. This choice is mainly due to the Cypress's CY7C68013 High Speed USB Peripheral chip, which implements USB physical and logical layer also it has an embedded 8051 microcontroller for increased system flexibility. Cypress also has available system software drivers, as well as a system SDK targeted to Visual Studio .NET programming tools, further reducing software development times. Given all this tools, the focus is to develop the FPGA interface and the client software (which is embedded in the Integrated Development & Management System). A system overview is visible in the figure below:

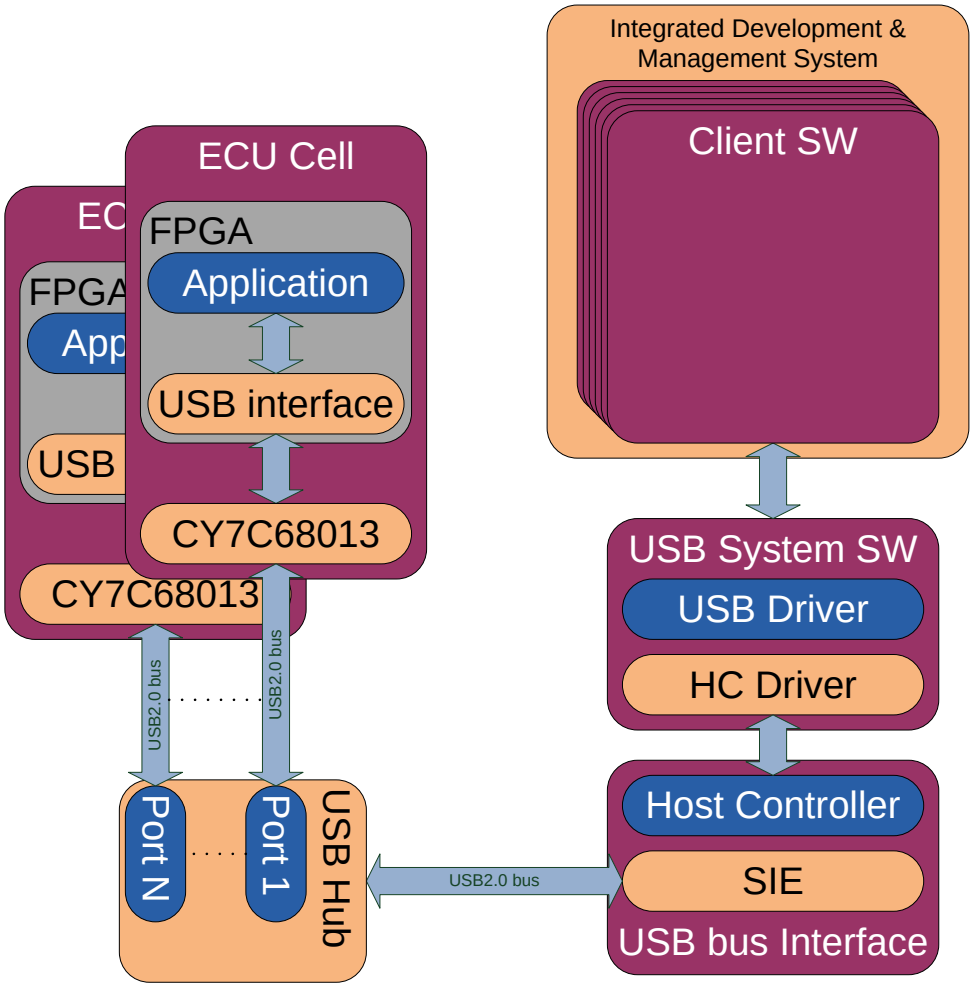


Figure 50 - Wired communications overview

It can be seen that in each ECU cell a CY7C68013 is present, interfacing with the FPGA application (ECU cell core) via a USB Interface specifically developed for this design.

All of the ECU Cells connect to a USB Hub, enabling access to the USB Host. On the end of the data path, we can see the Client software, embedded in the Integrated Development & Management System.

3.2.2.1 Cellular ECU wired communications scalability

As mentioned before, system scalability is required, as the Cellular ECU concept stands on system scalability in all aspects (processing power, program memory space, variable number, peripheral count and datalog capacity), as such, one of the objectives of the wired communications is to, not only enable a higher bandwidth access to the ECU cells, but also enable system scalability up to the USB high-speed mode full bandwidth, or as high as possible.

Scalability depends greatly on the data frame length sent trough USB, as well as the performance of the USB host and intermediate hubs, if applicable, and endpoints transfer mode. As there is not real-time requirement, all transfers are done in best effort. As such all endpoint transfers are configured to operate in bulk mode, as it offers the highest bandwidth possible (up to 52000 kBps for 512 byte packets). This allows adding as many ECU cells as needed to the USB bus, knowing that the USB Host will try to evenly distribute the available bandwidth for each ECU cell, in a best effort scheme, although there's no guaranteed bandwidth.

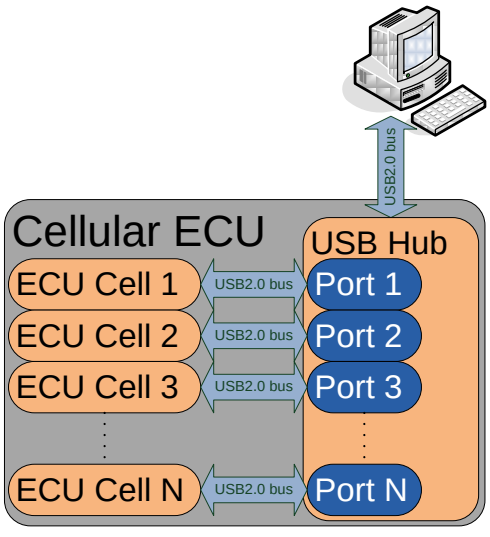


Figure 51 - Cellular ECU communication concept

The USB Hub is a part of the Cellular ECU, and is designed to accommodate all the ECU Cells, such as in Figure 51.

With the achieved peak bandwidth requirement (2,2MBps) the maximum number of ECU cells that is possible to add to a USB bus without bandwidth degradation would be approximately 26. But the fact that overhead bandwidth, latency introduced by the hub(s) and non ideal communication factors, such as bit-stuffing, roundtrip delays and ultimately electromagnetic noise, all add up to reduce the total number of possible ECU Cells in a USB bus without bandwidth degradation, at peak ECU Cell bandwidth

.

3.2.2.2 USB peripheral IC (CY7C68013)

The central component of the USB communications is the Cypress CY7C68013 chip, which implements the physical and logical layer of the USB stack. The Digilent's PmodUSB2 was used, as it provided a drop-in solution with little reconfiguration necessary.

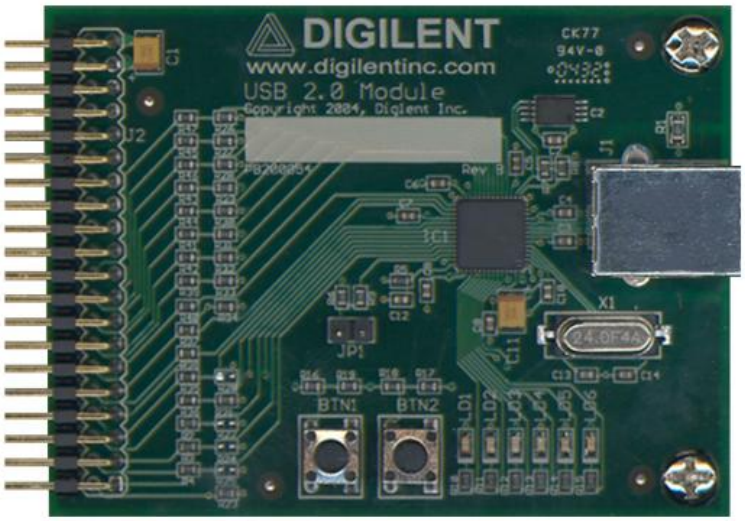


Figure 52 - Digilent PmodUSB2 (16)

CY7C68013 has an embedded 8051 microcontroller with enhanced performance, as most instructions execute in four clock cycles, and allows clock speeds up to 48 MHz.

It also provides for serial communication support (IIC and USART), GPIF interface (programmable finite state machine to interface with a broad variety of protocols such as ATA, parallel port and Utopia), up to 24 generic input / output pins, and one of the most important features, an direct access to the endpoint memory via a 4kB FIFO interface (8 to 16bit bus width, 5 to 30 MHz (60 MHz in 8 bit mode) clock speed), providing for extended USB packet buffering capabilities.

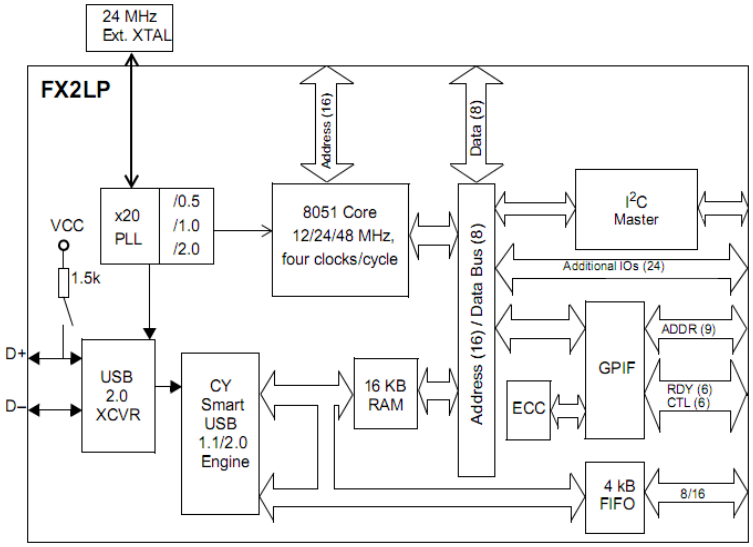


Figure 53 - Cypress CY7C68013 architecture (17)

Last but not least, the Cypress CY7C68013 also provides for an extended memory support, as is standard for 8051 microcontroller, in order to support larger program sizes and/or data memory requirements. This allows for up to 64kBytes program memory and 56kBytes data RAM.

3.2.2.3 USB interface

In order to simplify the endpoint access to the ECU cell, it was implemented a FIFO interface to the CY7C68013. The objective of this interface is to make the endpoint interface as simple as possible to the ECU cell.

As the ECU cell has a communication scheme that uses variable data width frames, packing those frames into an efficient USB frame sequence is of paramount importance if maximum performance is to be achieved.

Also the ECU cell has the need for parallel access to each USB endpoint; as such the provided FIFO interface by the CY7C68013 needed some kind of adaptation layer.

The solution was found using a second FIFO layer, implemented in the FPGA, in this case, one FIFO for each endpoint, allowing tackling the described problems.

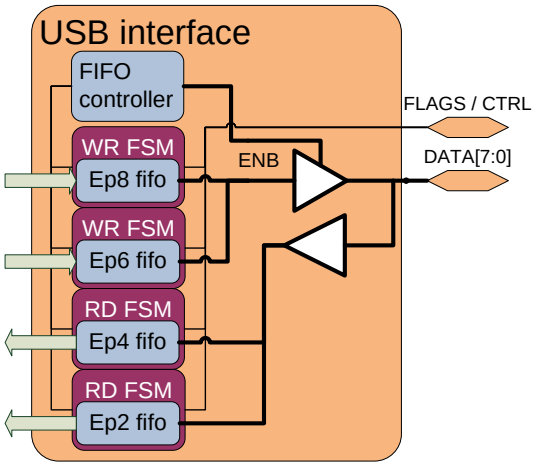


Figure 54 - USB Interface overview

The USB interface is built around four main components, each of them described below:

3.2.2.3.1 FIFO controller

This is a simple finite state machine, can be seen as an arbiter / sequencer, providing a timeslot for each endpoint to execute the data transfer.

3.2.2.3.2 Read FSM

A finite state machine, it controls data transfer from the Slave FIFO to the corresponding endpoint FIFO.

3.2.2.3.3 Write FSM

A finite state machine, it transfers data from the endpoint FIFO to the Slave FIFO. This finite state machine also manages the packet optimization algorithms, discussed further in this document.

3.2.2.3.4 Endpoint FIFO

Because of the multiple clock options that are in study for the application, it is advisable to build this block with a dual clock capability, one for the USB chip interface and the other for the ECU Cell FPGA core. These FIFO provide the interface between different clock domains and have the benefits of a large 2kB buffer that enable smarter data packetization.

The endpoint FIFO is generated with the Xilinx LogiCORE™ FIFO Generator, yielding a high performance and architecture optimized block.

3.2.2.4 USB client software

The USB client software is embedded in the Integrated Development & Management System (IDMS) and was developed to reduce overall software complexity, add greater flexibility and ease of use to the IDMS.

This software is part of an abstraction layer that enables communication to the ECU Cells trough several communication channels, from either local or remote systems.

The main feature of this block is that all communications are multithreaded, one thread per endpoint. As the USB bus is a pooled bus this enables the transaction of large data blocks without hogging the CPU even with low bandwidth, thus allowing a smoother user interface, as well as a software layer that uses the full power of recent multi-threaded, multi-core processors.

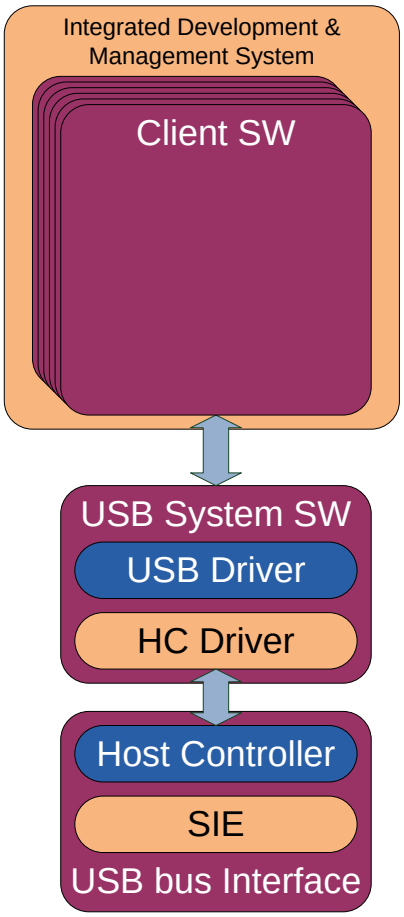


Figure 55 - Usb Host software stack

3.2.3 Wired communications details

3.2.3.1 USB peripheral IC (CY7C68013)

3.2.3.1.1 Chip signals

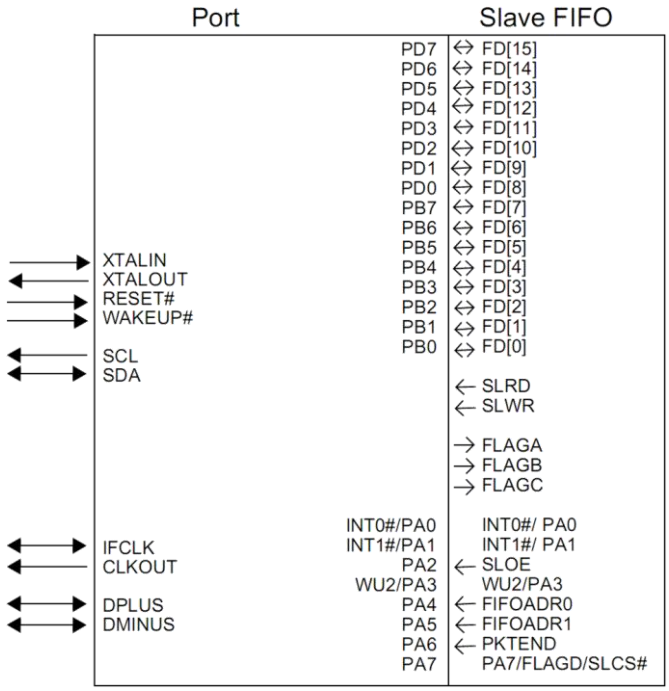


Figure 56 - CY7C68013 pins

The selected package was the 56 pin SSOP, as the extended features available in larger packages were not needed for this design.

PIN	Name	Type	Description
16	DMINUS	IO/Z	USB D- Signal
15	DPLUS	IO/Z	USB D+ Signal
49	RESET#	Input	Active LOW Reset
12	XTALIN	Input	Crystal Input
11	XTALOUT	Output	Crystal Output
5	CLKOUT	O/Z	8051 clock output
40	PA0/INT0#	IO/Z	IO / 8051 interrupt
41	PA1/INT1#	IO/Z	IO / 8051 interrupt
42	PA2/SLOE	IO/Z	IO / slave fifo output enable

43	PA3/WU2	IO/Z	IO / SB wakeup alternate source
44	PA4/FIFOADR0	IO/Z	IO / slave fifo address
45	PA5/FIFOADR1	IO/Z	IO / slave fifo address
46	PA6/PKTEND	IO/Z	IO / slave fifo commit packet
47	PA7/FLAGD/SLCS#	IO/Z	IO / slave fifo flag / slave fifo select
25	PB0/FD0	IO/Z	IO / slave fifo data bit
26	PB1/FD1	IO/Z	IO / slave fifo data bit
27	PB2/FD2	IO/Z	IO / slave fifo data bit
28	PB3/FD3	IO/Z	IO / slave fifo data bit
29	PB4/FD4	IO/Z	IO / slave fifo data bit
30	PB5/FD5	IO/Z	IO / slave fifo data bit
31	PB6/FD6	IO/Z	IO / slave fifo data bit
32	PB7/FD7	IO/Z	IO / slave fifo data bit
52	PD0/FD8	IO/Z	IO / slave fifo data bit
53	PD1/FD9	IO/Z	IO / slave fifo data bit
54	PD2/FD10	IO/Z	IO / slave fifo data bit
55	PD3/FD11	IO/Z	IO / slave fifo data bit
56	PD4/FD12	IO/Z	IO / slave fifo data bit
1	PD5/FD13	IO/Z	IO / slave fifo data bit
2	PD6/FD14	IO/Z	IO / slave fifo data bit
3	PD7/FD15	IO/Z	IO / slave fifo data bit
8	RDY0/SLRD	Input	GPIF input signal / slave fifo read strobe
9	RDY1/SLWR	Input	GPIF input signal / slave fifo write strobe
36	CTL0/FLAGA	O/Z	GPIF control output / slave fifo flag
37	CTL1/FLAGB	O/Z	GPIF control output / slave fifo flag
38	CTL2/FLAGC	O/Z	GPIF control output / slave fifo flag

20	IFCLK	IO/Z	slave fifo interface clock input/output
51	WAKEUP	Input	USB Wakeup
22	SCL	OD	IIC clock
23	SDA	OD	IIC data

Table 11 - CY7C68013 pin description

3.2.3.1.2 Embedded 8051 microcontroller

The CY7C68013 has an embedded 8051 microcontroller, allowing adding greater flexibility to the implemented system.

The current development phase of the design only requires that the embedded microcontroller is able to correctly configure the USB peripheral IC as a slave FIFO interface with the expected signal polarities and flag status. The microcontroller then idles and takes no part on data transactions to/from USB, yielding no bottleneck between the USB transceiver and the ECU cell's FPGA application.

Although it is possible to use this microprocessor built in ECC generator to further protect transmitted data between CY7C68013 and the FPGA, it was considered not to be necessary at this stage of prototyping because:

- The transmitted data already has built in EDC (CRC16)
- Developed application layer already implements data transmission handshaking, so data loss is minimized in the event of a random error.
- Low noise interface is easily achieved in final design as the CY7C68013 could be placed as physically close of the FPGA interface pins as needed due to the low overall ECU cell hardware complexity, yielding a reliable communication bus.
- The effort needed to implement ECC algorithms both in the FPGA and in the CY7C68013 is greatly oversized, given the real application benefits.
- It hampers the USB communications maximum achievable bandwidth, as the embedded 8051 needs to take part on the data transfer, adding interrupt latency and overhead to the transmitted data.

Nevertheless if the need to increase reliability between CY7C68013 and FPGA should arise (due to high noise environment present in automotive applications for instance), it would be simple enough to implement such ECC algorithms due to the reconfigurable nature of the FPGA and the currently underused capabilities of the CY7C68013.

The only important consideration to be taken if this solution is to be pursued further in the future is to implement extra handshaking and control signals between the ECU

Cell's FPGA and the CY7C68013, as all other aspects can be implemented by firmware changes.

3.2.3.1.3 Slave FIFO operation

3.2.3.1.3.1 Endpoint memory

CY7C68013 has the ability to operate in slave FIFO mode, providing a high speed, direct interface to the endpoint memory. This memory can be organized in 12 different arrangements, to enable flexibility to the numerous targeted systems.

The total endpoint memory is 4kByte and can be organized in the following configurations:

- 4x double buffered 512 bytes endpoints.
- 2x double buffered and 1x quad buffered 512 bytes endpoints.
- 2x double buffered 512 bytes and 1x double buffered 1024 bytes.
- 2x quad buffered 512 bytes endpoints.
- 1x quad buffered 512 bytes and 1x double buffered 1024 bytes endpoints.
- 2x double buffered 1024 bytes endpoints.
- 2x triple buffered and 1x double buffered 512 bytes endpoints.
- 1x triple buffered 1024 bytes and 1x double buffered 512 bytes endpoints.
- 1x quad buffered 1024 endpoint.

For the ECU cells, it was used the 4x double buffered 512 bytes configuration, as it provides four endpoints, so they can be configured as two IN endpoints and two OUT endpoints.

This configuration allows for grater system flexibility, while yielding a decent on chip buffering capability.

If system requirements do not need the four endpoint topology, unused ones could be aggregated to those currently in use either for greater packet length (yielding higher burst bandwidth) or for extended buffering (for reduced latency).

Along with the standard control endpoint 0, the CY7C68013 has a bidirectional endpoint 1 dedicated to the embedded 8051, allowing the future development of more complex communication topologies, if needed.

Endpoint 1 also enables CY7C6803 on-circuit configuration, allowing for firmware upgrades to be seamless, easy and fast.

3.2.3.1.3.2 Slave FIFO bus interface

For the ECU cells, the slave FIFO bus is configured as an 8 bit bus width, with a clock speed of 48MHz, sourced by the CY7C68013.

The bus has the following control signals (driven by the FPGA application):

- FIFOADR[1:0] – Endpoint FIFO address –selects the Slave FIFO to witch read / write operations are to be performed
- SLOE – Slave Output enable – controls the Slave FIFO tri-state output buffers
- SLRD – Slave Read – Strobe this pin to pop a Slave FIFO byte
- SLWR – Slave Write – Strobe this pin to push a byte into the Slave FIFO
- PKTEND – Packet End – Commits the partial packet (length < 512 bytes) to the USB SIE.
- The interface is also configured with the following flags (driven by CY7C68013):
- EF – Empty Flag – Indicates if Slave FIFO is empty
- FF – Full Flag – Indicates if Slave FIFO is full

3.2.3.1.3.3 Slave FIFO write

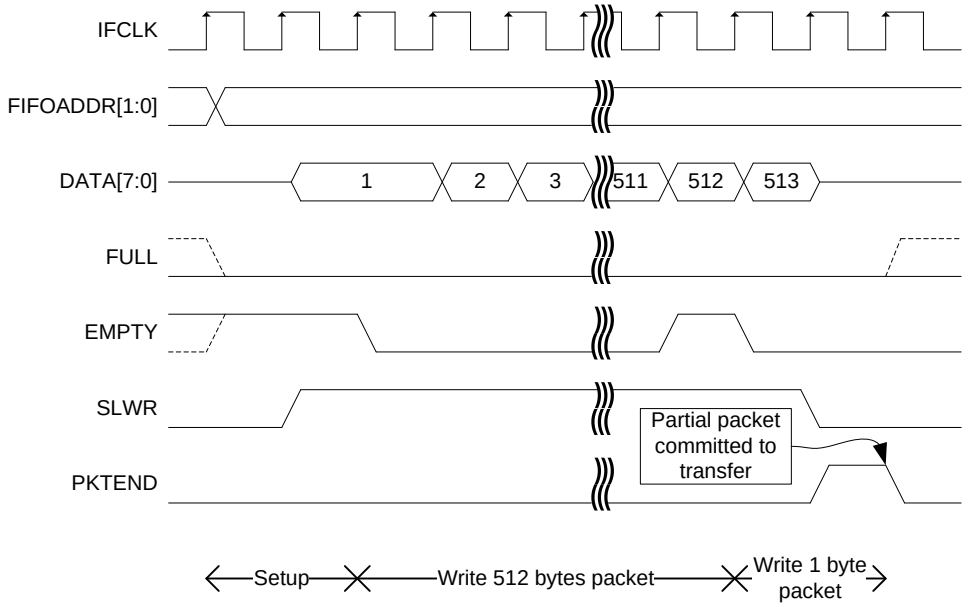


Figure 57 – Example Slave FIFO write

During write operations, the FULL flag is checked to determine if data can be pushed further to the Slave FIFO. In the example above a double packet is transmitted:

- The first one with 512 bytes of data is committed to transmission as the last byte is pushed into the Slave FIFO (auto commit function is enabled at chip power on sequence).
- The second packet is committed to transmission as soon as the PKTEND signal is driven (chip is configured to allow partial packets); in this case, right after the first byte is pushed into the Slave FIFO.

The selected configuration for the endpoints is double buffered 512 bytes endpoint, as such, if the first 512 byte packet is sent before the PKTEND is asserted, the FULL flag is not set, otherwise both levels of the endpoint memory will be used, causing the FULL flag to be asserted.

3.2.3.1.3.4 Slave FIFO read

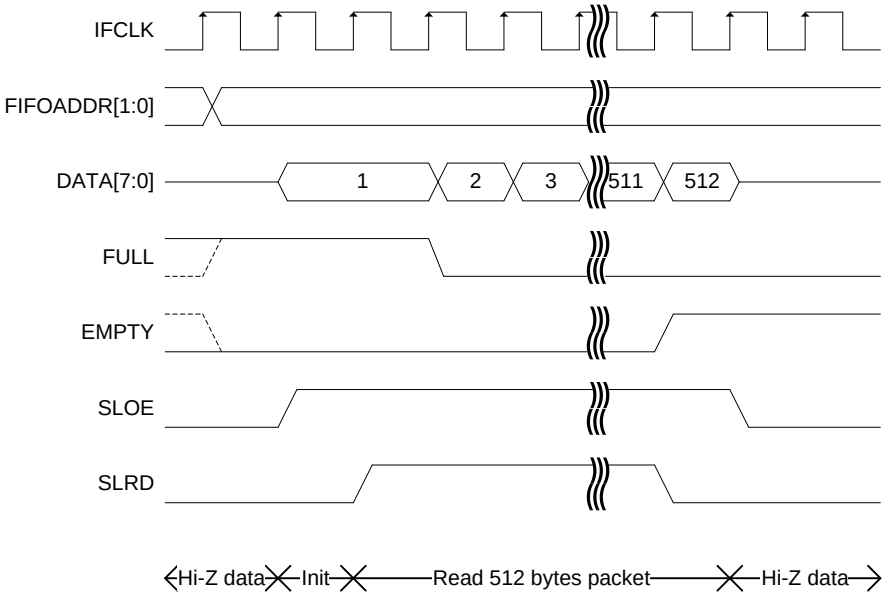


Figure 58 – Example Slave FIFO read

Similar to the write operations, the Slave FIFO read is performed monitoring the EMPTY flag. In the example above we see the transfer of a single 512 byte block:

- The data bus is High-Z as long as SLOE is reset.
- The read pointer is incremented by the SLRD.
- Read stops when the Slave FIFO is empty, by polling the EMPTY flag.
- After the transfer, SLOE is reset to put the data bus in High-Z state.

3.2.3.2 *USB interface – FIFO controller*

3.2.3.2.1 Interface signals

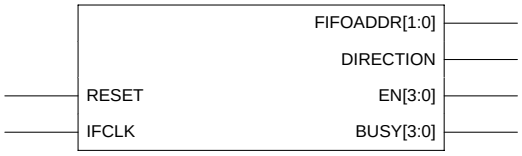


Figure 59 - FIFO controller interface signals

Signal Name	Width	Direction	Description
FIFOADDR	2	OUT	Slave FIFO endpoint select
DIRECTION	1	OUT	Tri-state buffer data bus direction
EN	4	OUT	Specific endpoint state machine enable
BUSY	4	IN	Specific endpoint state machine monitoring
IFCLK	1	IN	Slave fifo interface clock
RESET	1	IN	Synchronous reset

Table 12 - FIFO controller interface signals

3.2.3.2.2 Slave FIFO endpoint access arbitration

This block controls the sequence of operations in the Slave FIFO, in other words, the sequence of endpoints that are read / written. This is implemented with timeslots, allowing each endpoint a maximum duration and jumping to the next endpoint in the sequence if the current one has no activity.

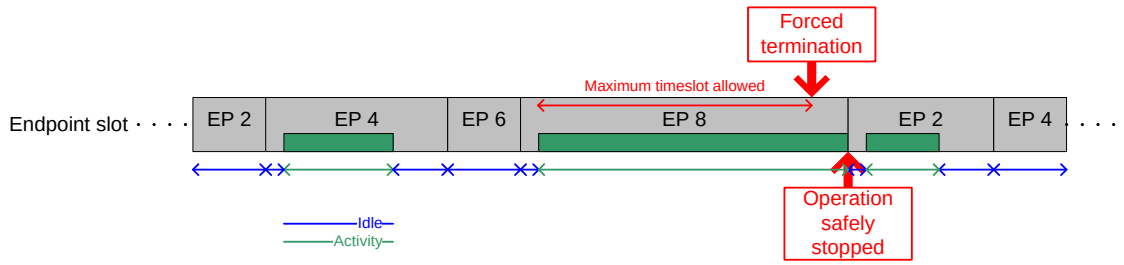


Figure 60 - FIFO controller sequence example

- It is visible in the previous figure that the endpoints are polled for activity during a brief time. If no transfer occurs during this time, the next endpoint in the sequence is activated.
- If for some reason the maximum timeslot is exceeded, a termination signal is sent to the corresponding state machine, allowing it to safely stop current operation and release the bus for the next endpoint transfer.
- This allows fair and reliable access to the endpoints, and can be configured anytime to any sequence and differentiated timeouts for each endpoint, to match the application needs.

3.2.3.3 USB interface – Slave FIFO read FSM

3.2.3.3.1 Interface signals

Figure 61 - Slave FIFO read FSM interface signals

Signal Name	Width	Direction	Description
RDEN	1	IN	Enable signal
BUSY	1	OUT	Busy signal
WEN	1	OUT	FIFO write enable signal
FULL	1	IN	FIFO full signal
EF	1	IN	Slave FIFO empty flag
SLOE	1	OUT	Slave FIFO output enable
SLRD	1	OUT	Slave FIFO read strobe
IFCLK	1	IN	Slave fifo interface clock
RESET	1	IN	Synchronous reset

Table 13 – Slave FIFO read FSM interface signals

3.2.3.3.2 Operation

The Slave FIFO read FSM controls the Slave FIFO – Endpoint FIFO datapath by controlling the SLOE and SLRD signals on the Slave FIFO interface and the WEN signal on the FPGA Endpoint FIFO.

The data transfer is enabled by the RDEN signal, and starts as soon as the Slave FIFO empty flag (EF) is not active and the internal FIFO is not full.

3.2.3.4 USB interface – Slave FIFO write FSM

3.2.3.4.1 Interface signals

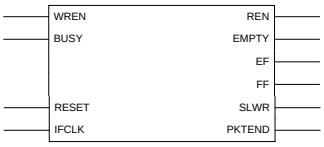


Figure 62 - Slave FIFO write FSM interface signals

Signal Name	Width	Direction	Description
WREN	1	IN	Enable signal
BUSY	1	OUT	Busy signal
REN	1	OUT	FIFO read enable signal
EMPTY	1	IN	FIFO full signal
EF	1	IN	Slave FIFO empty flag
FF	1	IN	Slave FIFO full flag
SLWR	1	OUT	Slave FIFO write strobe
PKTEND	1	OUT	Slave FIFO packet end strobe
IFCLK	1	IN	Slave FIFO interface clock
RESET	1	IN	Synchronous reset

Table 14 – Slave FIFO write FSM interface signals

3.2.3.4.2 Operation

The Slave FIFO write FSM is similar to the Slave FIFO read FSM as it controls the datapath between the Slave FIFO and the FPGA Endpoint FIFO. Its operation is enabled by the WREN signal.

3.2.3.4.3 Packet optimization

The CY7C68013 is configured to transmit 512 byte packets, although smaller packets are allowed, it could be highly inefficient to deviate from this packet length because USB throughput in Bulk mode is highly affected by the packet length.

Because the ECU Cell data frames have no fixed length, partial packets could dominate the transfers, hogging the USB bandwidth with packet overhead.

As a simple approach of simply committing a partial packet if no more data exists is not viable, another solution was implemented.

The solution found was to use the timeslot given by the FIFO controller to pack as many bytes to the endpoint memory as possible, minimizing the number of partial packets. When the timeslot ends, if there are remaining bytes, they are sent as a partial packet. This is done to ensure that all of the data is transmitted with a maximum guaranteed latency.

3.2.3.5 USB interface – Endpoint FIFO

3.2.3.5.1 Interface signals

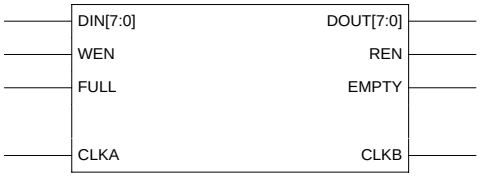


Figure 63 - Endpoint FIFO interface signals

Signal Name	Width	Direction	Description
DIN	8	IN	Enable signal
WEN	1	OUT	Busy signal
FULL	1	OUT	FIFO read enable signal
DOUT	8	IN	FIFO full signal
REN	1	IN	Slave FIFO empty flag
EMPTY	1	IN	Slave FIFO full flag
CLKA	1	IN	Slave FIFO interface clock
CLKB	1	IN	Synchronous reset

Table 15 – Endpoint FIFO interface signals

3.2.3.5.2 Details

The endpoint FIFO is generated using the Xilinx LogiCORE™ FIFO Generator, this is a standard FIFO with dual clock source, configured to have a 2kByte with 8 bit bus width on both domains.

The same architecture is used on all endpoint buffers.

3.2.3.6 Client software

3.2.3.6.1 Organization

The client software manages the communications between the application (IDMS) and the USB Device drivers. As stated before, in order to make the user interface more fluid when transferring large amounts of data, a multithreaded approach was used, using the .NET environment provided by the tool Visual Studio 2005 tool suite in Visual C#

The multithreaded approach allows the application to take full advantage of the new processor generation which has been growing in multi-thread capability by increasing the number of processor cores.

A static class was implemented that manages the USB device insertion and removal automatically, as well as message reception and transmission. An event based interface was used.

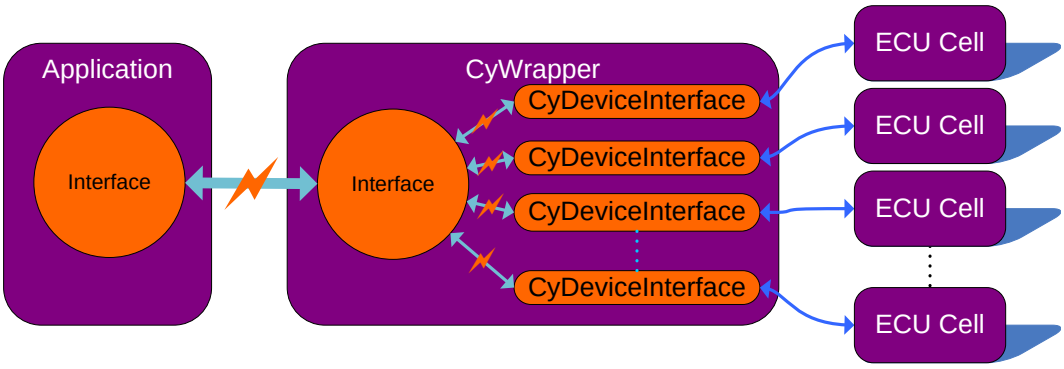


Figure 64 - Client software organization

The available events are:

- Device added – This event is called every time a new ECU Cell is plugged to the USB bus, detailed information is provided by the event upon call.
- Device removed – This event is called when an ECU Cell is removed from the USB bus. The ECU Cell information is provided upon event call.
- Bytes received – Event called when the committed data is successfully received via USB Bus.
- Bytes sent – An event that is called when data is sent over the USB Bus.
- Work terminated – This event is called when all of the threads associated to a specific device's endpoints are terminated.

3.2.4 USB Interface operation

3.2.4.1 Summary

The USB Interface is a self contained block, allowing a simple drop in solution to any design, although this has been optimized for the ECU Cell core, nevertheless in this sections it is described how to initialize and use this block

3.2.4.2 Interface signals

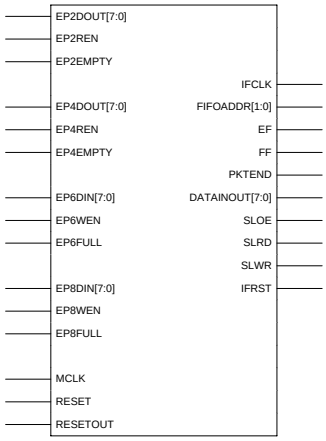


Figure 65 - USB Interface signals

Signal Name	Width	Direction	Description
EP2OUT	8	OUT	Endpoint 2 DIDO data output
EP2REN	1	IN	Endpoint 2 FIFO read strobe
EP2EMPTY	1	OUT	Endpoint 2 FIFO empty flag
EP4OUT	8	OUT	Endpoint 4 DIDO data output
EP4REN	1	IN	Endpoint 4 FIFO read strobe
EP4EMPTY	1	OUT	Endpoint 4 FIFO empty flag
EP6IN	8	IN	Endpoint 6 DIDO data input
EP6WEN	1	IN	Endpoint 6 FIFO write strobe
EP6FULL	1	OUT	Endpoint 6 FIFO full flag

EP8IN	8	IN	Endpoint 8 DIDO data input
EP8WEN	1	IN	Endpoint 8 FIFO write strobe
EP8FULL	1	OUT	Endpoint 8 FIFO full flag
MCLK	1	IN	FPGA master clock input
RESET	1	IN	Reset input
RESETOUT	1	OUT	Reset output
EF	1	IN	Slave FIFO empty flag
FF	1	IN	Slave FIFO full flag
PKTEND	1	OUT	Slave FIFO packet end strobe
DATAINOUT	8	INOUT	Slave FIFO data bus
SLOE	1	OUT	Slave FIFO output enable
SLRD	1	OUT	Slave FIFO read strobe
SLWR	1	OUT	Slave FIFO write strobe
IFRST	1	OUT	CY7C68013 reset

Table 16 - USB Interface signals

3.2.4.3 Initialization

No initialization is needed. After the release of the reset signal, the USB is ready to use when the RESETOUT goes low.

3.2.4.4 Endpoint read

Endpoint 2 and 4 direction is configured as OUT. The read operations are performed on the corresponding Endpoint FIFOs on the FPGA, so direct action is not needed.

3.2.4.5 Endpoint write

Endpoint 6 and 8 direction is configured as IN. As the endpoint 2 and 4 the write operations are performed on the corresponding Endpoint FIFOs on the FPGA, so direct action is not needed.

3.2.5 USB Interface detailed schematic

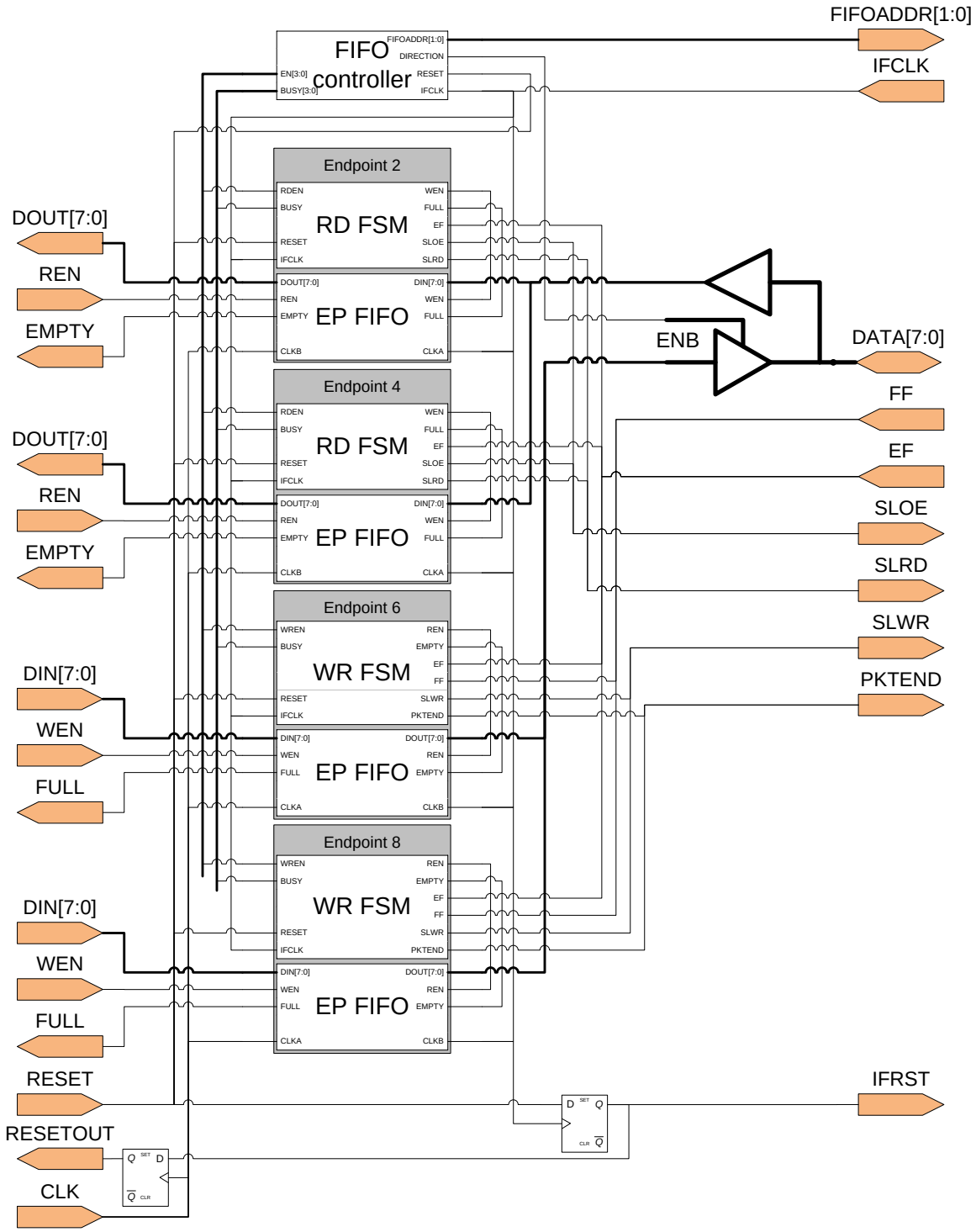


Figure 66 - USB Interface detailed schematic

The main components are represented on the figure above, with the Cypress CY7C68013 interface signals at the right side and the ECU Cell core interface at the left side.

3.3 Wireless communications

3.3.1 nRF24L01 Product Specification

The nRF24L01 is developed by Nordic Semiconductor ASA, a low-cost, high-performance RF Transceiver with an embedded baseband protocol engine – Enhanced ShockBurst™. Designed to operate in the ISM band (2400 MHz – 2483.5 MHz), it requires few external components and interfaces with the application trough a SPI bus. The device already incorporates high PSRR internal supplies, allowing operation from a wide and unregulated voltage power sources.

Figure 67 - nRF24L01 reference modules (18)

It uses GFSK modulation at the radio frontend, and has configurable frequency, output power and air data rate. It also allows configuring a multitude of baseband protocol options for application fine-tuning or even baseband protocol override, whenever needed.

3.3.1.1 Radio features

- 2.4GHz ISM band operation
- 126 RF channels
- Common RX and TX pins at radio frontend.
- GFSK modulation
- 1Mbps / 2Mbps air data rates with 1 MHz / 2 MHz non-overlapping channel spacing
- Programmable output power: 0, -6, -12 or -18 dBm
    - 11.3 mA consumption at 0 dBm output power
- Integrated channel filters
    - 12.3mA consumption at 2MBps Rx mode

- -82 dBm sensitivity at 2 Mbps
- -85 dBm sensitivity at 1 Mbps
- Programmable LNA gain
- Fully integrated RF synthesizer
- No external loop filter, VCO, varactor diode or resonator required
- Accepts ±60ppm 16 MHz crystal

3.3.1.2 Baseband protocol features

- 1 to 32 bytes dynamic payload length
- Automatic packet handling (CRC16 or CRC23 generation and checking, preamble generation, address decoding and transmittal)
- Automatic packet transaction handling (Automatic acknowledgement transmittal)
- 6 data pipe MultiCeiver™ for up to 1:6 star networks

3.3.2 Wireless communications overview

This application, an ECU, generally relies on local datalogging for offline data analysis. Because of the increasing ECU functions complexity, variable count and higher sample rates, a flexible yet powerful solution was needed.

The idea of developing a wireless datalogger was born. This kind of solution makes swapping datalogger in runtime easy as the unit could be self powered, thus neither connectors nor complex retention devices would be needed.

Another extended capability of wireless communications is to enable remote system control and monitoring, similar to the one provided by the wired communications, given that there's enough bandwidth.

Given the scalable nature of the Cellular ECU, a scalable wireless solution was sought. The main architecture was defined as follows:

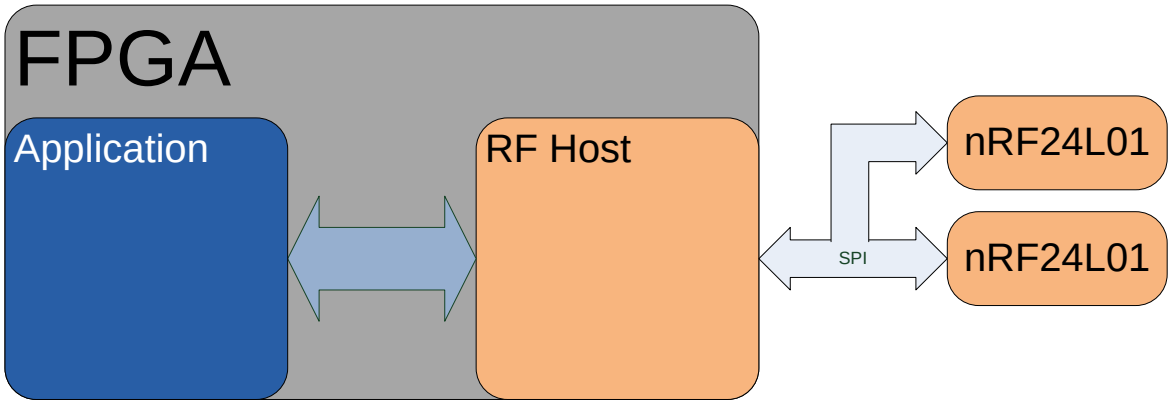


Figure 68 - Wireless communications overview

The solution was built using dual Nordic's nRF24L01 chip, along with a standard SPI bus connecting the chips to the FPGA.

The RF host has the following main components:

1. Payload cache
2. Payload flags
3. Rf Host:
    a. SPI master
    b. Command parser
    c. Setup FSM
    d. Configure FSM
    e. Receive FSM
    f. Transmit FSM

3.3.2.1 nRF24L01

The nRF24L01 is a transceiver chip made by Nordic Semiconductor that enables digital communications in the ISM band (2.4GHz), using GFSK modulation.

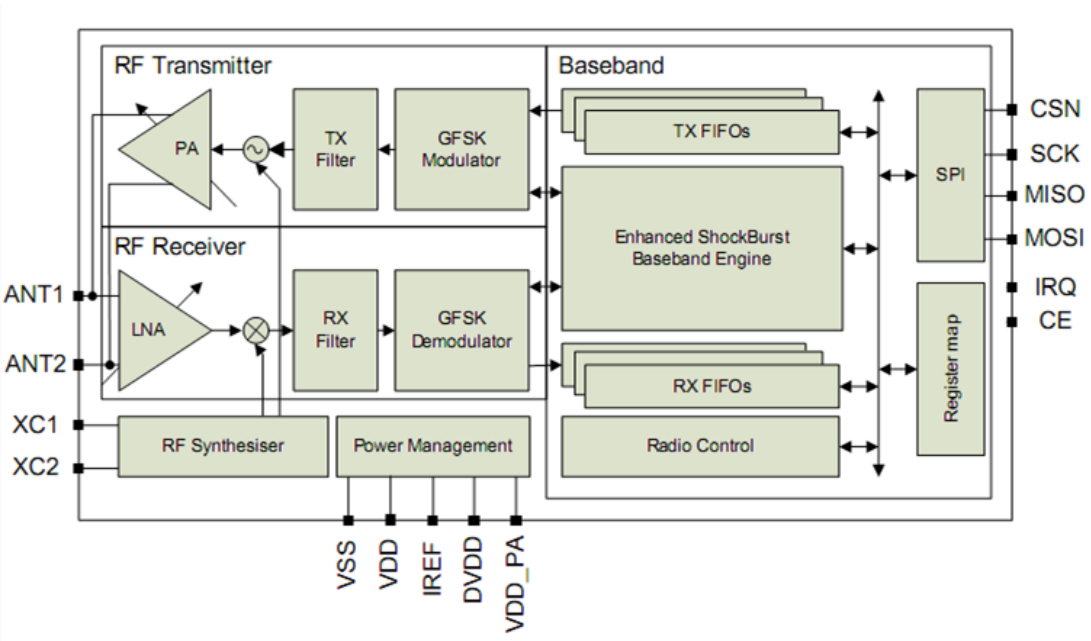


Figure 69 - nRF24L01 diagram (12)

This chip encapsulates all the RF electronics needed to transmission and reception, yielding a very low external component count.

It implements features such as automatic packet retransmission, header and CRC automatic generation and verification and fast channel switching with burst speeds up to 2Mbps.

The interface to the FPGA is done via SPI with a operating clock speeds up to 8 MHz, allowing more than enough bandwidth to cope with the RF link bandwidth.

As an interesting feature, this chip provides the VDD_PA signal, which is the internal PA 1.8V power supply and is only active during transmission, allowing easy interface with an external RF power amplifier, for increased range.

3.3.2.2 Payload cache

The payload cache is part of the RF Host and implements a temporary memory location where all packets to be sent or received are stored. It also includes the configuration memory space, allowing the ECU Cell core to configure the RF chips and enable for example frequency hopping algorithms, broadcast messages and device discovery to be implemented.

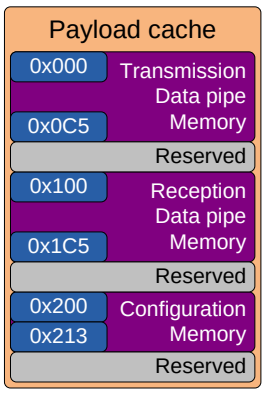


Figure 70 - Payload cache

3.3.2.3 Payload flags

The payload flags are a bitmap of every data pipe cache status; an empty data pipe is represented by a logical '0'.

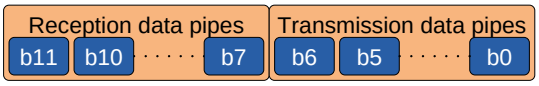


Figure 71 - Bit assignment

3.3.2.4 RF processor

The RF processor has the task to implement the read / write packet functions, as well the configuration at power on / reset and user driven configuration changes.

Each function is implemented as an independent FSM, allowing maximum performance in the most common operations, and the possibility to add in the future extra hardware implemented functions such as frequency hopping algorithms.

Each hardware implemented function is described further below.

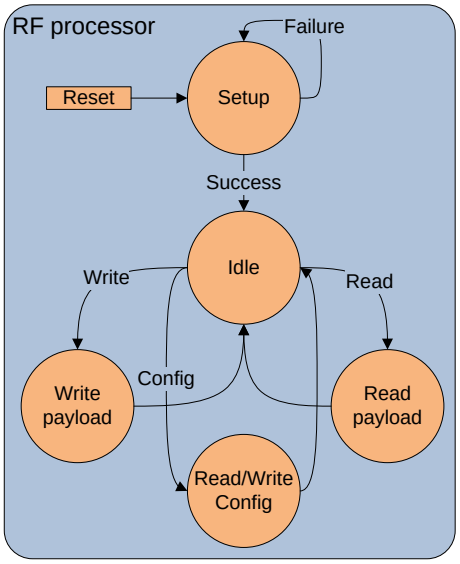


Figure 72 - RF processor overview

3.3.2.5 SPI master

The SPI master is part of the RF processor and implements the physical interface with the dual Nordic nRF24L01 solution.

This module only controls the SCK, MOSI and MISO signals, the chip enable and chip select signals are controlled by other blocks in the RF Host.

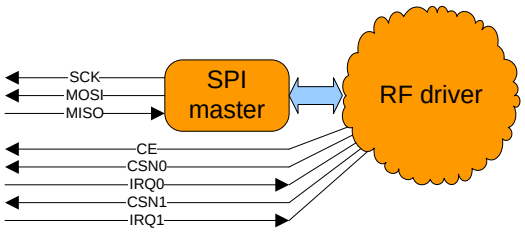


Figure 73 - SPI master overview

3.3.2.6 Command parser

This is part of the RF Processor and builds the data frames to be transmitted to each nRF24L01 chip, processing the command responses, interfacing with the SPI master.

Any hardware function implemented interfaces with this block, allowing additional functions to be easily implemented.

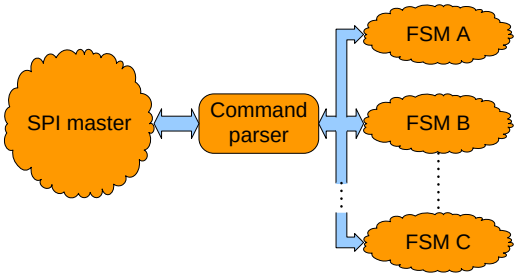


Figure 74 - Command parser overview

3.3.2.7 Setup FSM

This FSM resets both Nordic's nRF24L01 registers to predetermined default values after system power-on or reset. The predefined values are loaded on a ROM embedded on the FPGA design.

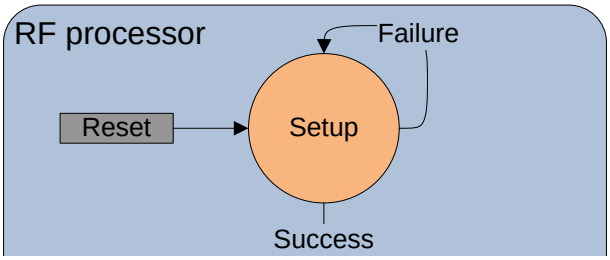


Figure 75 - Setup FSM overview

3.3.2.8 Configure FSM

The payload cache also contains a shadow of the Nordic's nRF24L01; this FSM is called whenever this shadow registers need to be refreshed, or to reflect some change in those shadow registers into the nRF24L01 configuration registers.

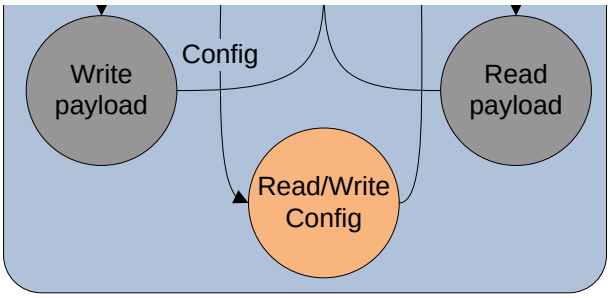


Figure 76 - Configure FSM

3.3.2.9 Receive FSM

The packet reception triggers this FSM to transfer the newly received payload from the nRF24L01 chip internal memory into the Payload Cache.

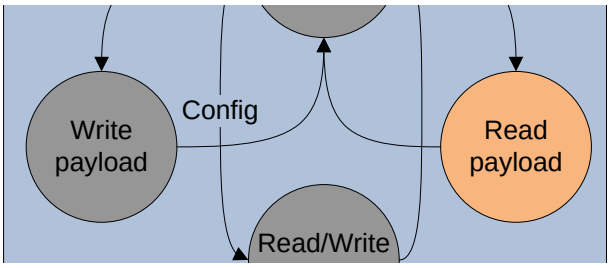


Figure 77 - Receive FSM overview

3.3.2.10 Transmit FSM

Whenever a new packet is ready to be transmitted, this FSM is triggered to transfer the corresponding data from the Payload Cache into the nRF24L01 chip, triggering a new transmission.

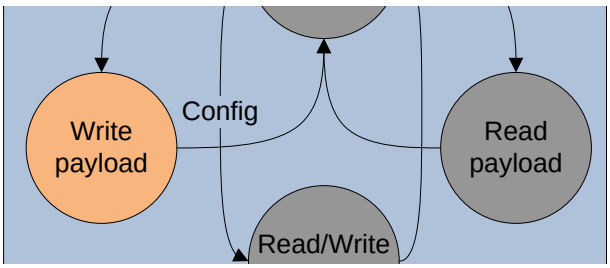


Figure 78 - Transmit FSM overview

3.3.2.11 Arbiter FSM

All of the above FSM's are controlled by this FSM, which acts as a sequencer, triggering the correct FSM into operation whenever a task is to be performed.

Also the operation priority and bus arbitration is implemented on this block, as there it is the need to control the bus access to prevent a function from hogging the bus for too long.

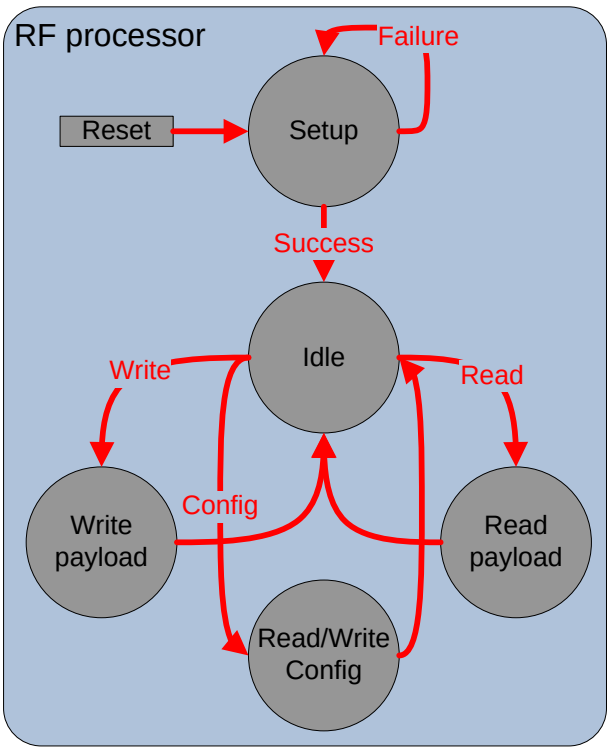


Figure 79 - Arbiter FSM overview

3.3.3 Wireless communications details

3.3.3.1 nRF24L01

3.3.3.1.1 Chip signals

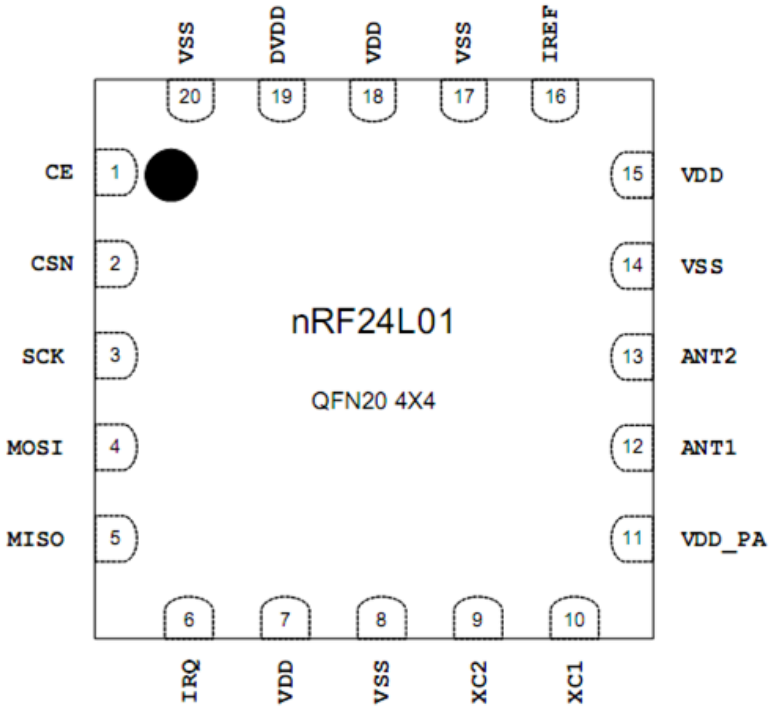


Figure 80 - nRF24L01 chip signals (12)

PIN	Name	Type	Description
1	CE	Digital Input	Chip Enable Activates RX or TX mode
2	CSN	Digital Input	SPI Chip Select
3	SCK	Digital Input	SPI Clock
4	MOSI	Digital Input	SPI Slave Data Input
5	MISO	Digital Output	SPI Slave Data Output, with tri-state option
6	IRQ	Digital Output	Maskable interrupt pin. Active low
7	VDD	Power	Power Supply
8	VSS	Power	Ground
9	XC2	Analog Output	Crystal Pin 2
10	XC1	Analog Input	Crystal Pin 1

11	VDD_PA	Power Output	Power Supply Output for the internal nRF24L01 PA
12	ANT1	RF	Antenna interface 1
13	ANT2	RF	Antenna interface 2
14	VSS	Power	Ground
15	VDD	Power	Power Supply
16	IREF	Analog Input	Reference current. Connect a 22kΩ resistor to ground.
17	VSS	Power	Ground
18	VDD	Power	Power Supply
19	DVDD	Power Output	Internal digital supply output for de-coupling purposes.
20	VSS	Power	Ground

Table 17 - CY7C68013 pin description

3.3.3.1.2 Enhanced ShockBurst™

Enhanced ShockBurst™ is a feature of this chip, implementing a data link layer. Some of the features of this data link layer are:

- Automatic packet assembly
- Dynamic payload length
- Automatic transmission handling, with automatic packet re-transmission and acknowledgements.
- Address based packet transmission and MultiCeiver™ capability.

By integrating these functionalities, the nRF24L01 shifts most of the complexity of the physical layer and link layer to itself, allowing focusing in higher abstraction functions on the host and cutting short the development time.

Also, by having an address based communication, it is relatively simple to establish point-to-point communications networks, or even a star communication network.

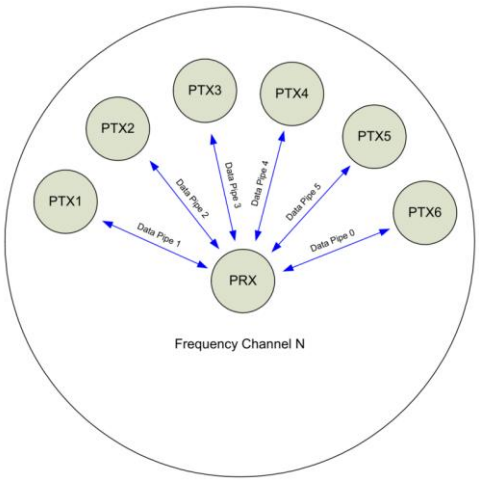


Figure 81 - Star network example (12)

3.3.3.2 Payload cache

3.3.3.2.1 Interface signals

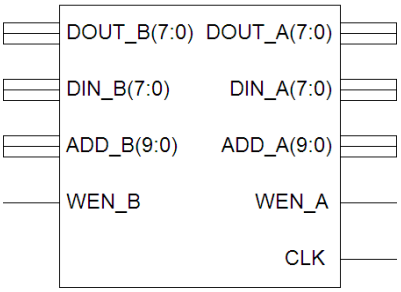


Figure 82 – Payload cache Interface signals

Signal Name	Width	Direction	Description
DOUT_A	8	OUT	Port A data output
DIN_A	8	IN	Port A data input
ADD_A	10	IN	Port A address input
WEN_A	1	IN	Port A write enable
DOUT_B	8	OUT	Port B data output
DIN_B	8	IN	Port B data input
ADD_B	10	IN	Port B address input
WEN_B	1	IN	Port B write enable
CLK	1	IN	Clock

Table 18 - Payload cache signals

3.3.3.2.2 Memory mapping

Start Address	Description	Length (bytes)
0x000	Tx Payload pipe 0	32
0x020	Tx Payload pipe 1	32
0x040	Tx Payload pipe 2	32
0x060	Tx Payload pipe 3	32
0x080	Tx Payload pipe 4	32

0x0A0	Tx Payload pipe 5	32
0x0C0	Tx Payload pipe 0 length	1
0x0C1	Tx Payload pipe 1 length	1
0x0C2	Tx Payload pipe 2 length	1
0x0C3	Tx Payload pipe 3 length	1
0x0C4	Tx Payload pipe 4 length	1
0x0C5	Tx Payload pipe 5 length	1
0x0C6	Reserved	58
0x100	Rx Payload pipe 0	32
0x120	Rx Payload pipe 1	32
0x140	Rx Payload pipe 2	32
0x160	Rx Payload pipe 3	32
0x180	Rx Payload pipe 4	32
0x1A0	Rx Payload pipe 5	32
0x1C0	Rx Payload pipe 0 length	1
0x1C1	Rx Payload pipe 1 length	1
0x1C2	Rx Payload 2 pipe length	1
0x1C3	Rx Payload pipe 3 length	1
0x1C4	Rx Payload pipe 4 length	1
0x1C5	Rx Payload pipe 5 length	1
0x1C6	Reserved	58
0x200	Pipe enable bitmap	1
0x201	Address width	1
0x202	Automatic retry configuration	1
0x203	Transmit channel select	1

0x204	Receive channel select	1
0x205	RF parameters setup	1
0x206	Pipe 0 Address	5
0x20B	Pipe 1 Address	5
0x210	Pipe 2 Address	1
0x211	Pipe 3 Address	1
0x212	Pipe 4 Address	1
0x213	Pipe 5 Address	1
0x214	Reserved	492

Table 19 - Payload cache memory map

3.3.3.3 Payload flags

3.3.3.3.1 Interface signals

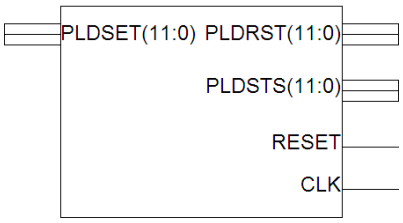


Figure 83 - Payload flags signals

Signal Name	Width	Direction	Description
PLDSET	12	IN	Flag set bitmap
PLDRST	12	IN	Flag reset bitmap
PLDSTS	12	OUT	Flag bitmap
RESET	1	IN	Reset
CLK	1	IN	Clock

Table 20 Payload flags signals

3.3.3.3.2 Bit mapping

Bit	Description	'0'	'1'

0	Transmit pipe 0 status	New payload to send	Payload sent
1	Transmit pipe 1 status	New payload to send	Payload sent
2	Transmit pipe 2 status	New payload to send	Payload sent
3	Transmit pipe 3 status	New payload to send	Payload sent
4	Transmit pipe 4 status	New payload to send	Payload sent
5	Transmit pipe 5 status	New payload to send	Payload sent
6	Receive pipe 0 status	Empty	New payload received
7	Receive pipe 1 status	Empty	New payload received
8	Receive pipe 2 status	Empty	New payload received
9	Receive pipe 3 status	Empty	New payload received
10	Receive pipe 4 status	Empty	New payload received
11	Receive pipe 5 status	Empty	New payload received

Table 21 - Payload flags bitmap

3.3.3.4 RF processor

3.3.3.4.1 Interface signals

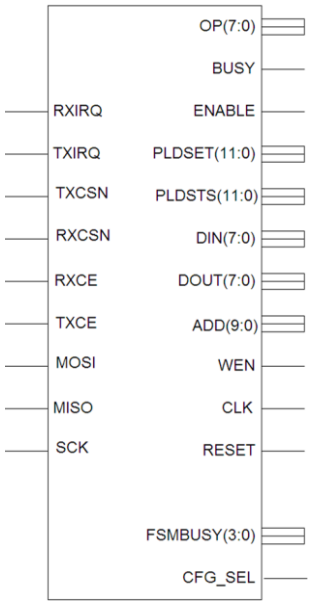


Figure 84 - RF Processor signals

Signal Name	Width	Direction	Description
RXIRQ	1	IN	Receive chip interrupt signal
TXIRQ	1	IN	Transmit chip interrupt signal
TXCSN	1	OUT	Transmit chip select signal
RXCSN	1	OUT	Receive chip select signal
RXCE	1	OUT	Receive chip enable signal
TXCE	1	OUT	Transmit chip enable signal
MOSI	1	OUT	SPI Master Output, Slave Input
MISO	1	IN	SPI Master Input, Slave Output
SCK	1	OUT	SPI clock
OP	8	IN	Operation mode
BUSY	1	OUT	Busy signal

ENABLE	1	IN	Enable signal
PLDSET	12	OUT	Payload set bitmap
PLDSTS	12	IN	Payload status bitmap
DIN	8	IN	Payload cache data in
DOUT	8	OUT	Payload cache data out
ADD	10	OUT	Payload cache address
WEN	1	OUT	Payload cache write enable
CLK	1	IN	Master clock
RESET	1	IN	Reset signal
FSMBUSY	4	OUT	FSM busy signal – For debug purposes
CFG_SEL	1	IN	Configuration select signal

Table 22 - RF processor signals

3.3.3.4.2 Operation modes

3.3.3.4.2.1 Packet transmission

To enable standard data packet transmission, the OP vector must be loaded with 0x00, and the enable signal must be active. During this operation mode, the reception and transmission pipes are polled, and data is transferred from/into the payload cache as needed.

3.3.3.4.2.2 Configuration readout

To readout the configuration of reception and transmission devices, the OP vector is loaded with 0x02, and the enable signal must be active to perform this operation.

After the transition from high to low of the busy signal, the configuration registers mapped on the payload cache are updated, allowing access to specific device configuration.

3.3.3.4.2.3 Configuration write

When writing a new configuration, it is possible to choose either updating both devices, and only one of them (reception or transmission). To reflect the changes in the configuration registers mapped on the payload cache on both devices the OP vector is loaded with the value 0x03, followed by an active enable signal. Operation is in progress while the busy signal is active, and upon transition from high to low of the busy signal the operation completes.

In some cases it is desirable to update only one of the devices with the new configuration, in such event it is possible to select the target device (transmission or reception) simply by loading the OP vector accordingly: 0x04 for reception configuration 0x05 for transmission configuration.

OP	Description
0x05	Configure Tx device channel
0x04	Configure Rx device channel
0x03	Write all configuration to devices
0x02	Read all configuration from devices
0x00	Normal Rx/Tx operation

Table 23 - RF processor operation modes

3.3.3.5 SPI master

3.3.3.5.1 Interface signals

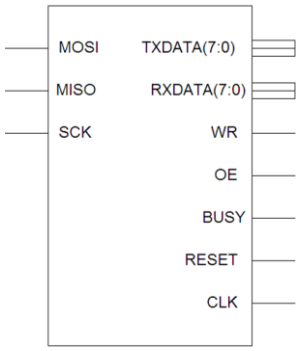


Figure 85 - SPI master interface signals

Signal Name	Width	Direction	Description
MOSI	1	OUT	SPI Master Output, Slave Input
MISO	1	IN	SPI Master Input, Slave Output
SCK	1	OUT	SPI clock
TXDATA	8	IN	SPI data to transmit
RXDATA	8	OUT	SPI received data
WR	1	IN	SPI start of transmission signal
OE	1	IN	RXDATA output enable signal
BUSY	1	OUT	SPI busy signal
RESET	1	IN	Reset
CLK	1	IN	SPI Clock reference

Table 24 – SPI master signals

3.3.3.5.2 Operation

This SPI master is designed to match signal polarity and phase of the nRF24L01 chips, also the SPI clock is generated in this block.

As the SCK (SPI clock) can be much lower than the Cell core clock, the SCK generation as well as MOSI driving and MISO sampling must be performed in a slightly different way than usual to avoid unnecessary dedicated clock routing resource consumption.

If standard techniques were used, such as implementing a clock divisor to generate the SCK and drive a finite state machine with this signal, additional clock routing resources would be used, leading to an inefficient design, to avoid this, a finite state machine driven directly from the Cell clock and with dual clock enable signals was implemented.

To drive these dual clock enable, a clock enable generator with a 0º and a 180º output was implemented. This clock enable generator creates a pulse with programmable period (an integer multiple of the Cell core clock cycle) and one Cell clock cycle width, and also creates a similar clock enable pulse with 180º phase.

When transmitting, at each 0º clock enable pulse, the SPI FSM shifts in the MISO signal and drives the SCK high and on every 180º clock enable pulse the SPI FSM shifts out the MOSI line and drives the SCK signal low, allowing the SPI FSM to run directly from the Cell clock, yielding lower clock routing resources consumption.

3.3.3.6 Command parser

3.3.3.6.1 Interface signals

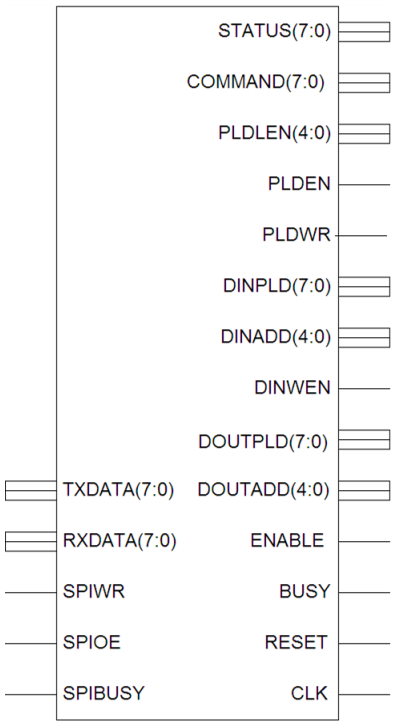


Figure 86 - Command parser interface signals

Signal Name	Width	Direction	Description
TXDATA	8	OUT	SPI data to transmit
RXDATA	8	IN	SPI data received
SPIWR	1	OUT	SPI transmit enable
SPIOE	1	OUT	SPI RXDATA output enable
SPIBUSY	1	IN	SPI busy signal
STATUS	8	OUT	Last received status
PLDLEN	8	IN	Payload length
PLDEN	5	IN	Payload enable
PLDWR	1	IN	‘1’ - Write payload, ‘0’ – Read payload
DINPLD	8	OUT	Received payload data

DINADD	5	OUT	Received payload address
DINWEN	1	OUT	Received payload write enable
DOUTPLD	8	IN	Payload data to transmit
DOUTADD	5	IN	Payload address to transmit
ENABLE	1	IN	Enable signal
BUSY	1	OUT	Busy signal
RESET	1	IN	Master reset
CLK	1	IN	Cell clock

Table 25 – SPI master signals

3.3.3.7 Setup FSM

3.3.3.7.1 Interface signals

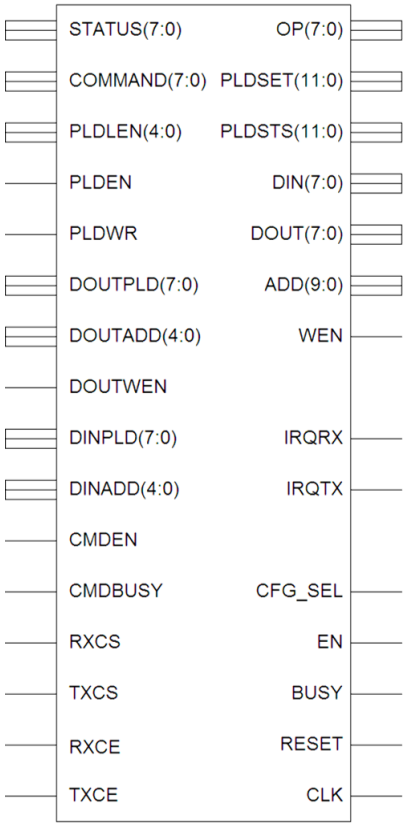


Figure 87 - Setup FSM interface signals

Signal Name	Width	Direction	Description
STATUS	8	OUT	Last command status
COMMAND	8	OUT	Command to send
PLDLEN	5	OUT	Payload length
PLDEN	1	OUT	Payload enable signal
PLDWR	1	OUT	Send or receive payload: ‘1’ Write; ‘0’ Read
DOUTPLD	8	OUT	Output payload data
DOUTADD	5	OUT	Output payload address
DOUTWEN	1	OUT	Output payload write enable

DINPLD	8	IN	Input payload data
DINADD	5	IN	Input payload address
CMDEN	1	OUT	Command enable
CMDBUSY	1	IN	Command busy
RXCS	1	OUT	Receive device chip select
TXCS	1	OUT	Transmit device chip select
RXCE	1	OUT	Receive device chip enable
TXCE	1	OUT	Transmit device chip enable
OP	8	IN	Operation mode
PLDSET	12	OUT	Payload flags set bitmap
PLDSTS	12	IN	Payload flags status
DIN	8	IN	Payload cache data input
DOUT	8	OUT	Payload cache data output
ADD	10	OUT	Payload cache address
WEN	1	OUT	Payload cache write enable
IRQRX	1	IN	Receive device interrupt
IRQTX	1	IN	Transmit device interrupt
CFG_SEL	1	IN	Default configuration select
EN	1	IN	Enable signal
BUSY	1	OUT	Busy signal
RESET	1	IN	Master reset
CLK	1	IN	Cell clock

Table 26 – Setup FSM signals

3.3.3.8 Configure FSM

3.3.3.8.1 Interface signals

Identical to the signals described in the Setup FSM.

3.3.3.9 Receive FSM

3.3.3.9.1 Interface signals

Identical to the signals described in the Setup FSM.

3.3.3.10 Transmit FSM

3.3.3.10.1 Interface signals

Identical to the signals described in the Setup FSM.

3.3.3.11 Arbiter FSM

3.3.3.11.1 Interface signals

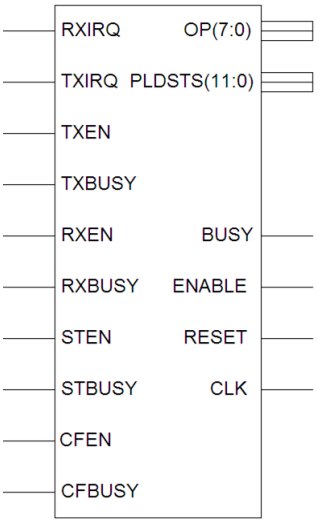


Figure 88 - Arbiter FSM interface signals

3.3.3.11.2 Operation

After a power on reset or after a system reset, the arbiter FSM starts the configuration sequence, as the nRF24L01 status is unknown. As such it configures all of the registers with the default values defined within the Setup FSM and flushes all of the receive and transmission FIFOs on both devices to ensure that operation always resumes from a predetermined state.

It was possible to revert both nRF24L01 chips into a predetermined state by simply cycling it's power, but in that case additional hardware would be necessary and the FPGA's resource savings would not be as much as they could be on the first glance

because a timing mechanism would still be necessary to guarantee a minimum time for the power cycle to clear both nRF24L01 registers.

As such this method was preferred to the power cycle as it added the flexibility to pre-program a fixed configuration that would enable simple point-to-point communications without extra operations by the Cell core, allowing debugging the system and testing this block separately from the rest of the Cell core.

3.3.4 Wireless communications operation

3.3.4.1 Summary

The following section describes the operation of the RF Host.

3.3.4.1.1 Interface signals

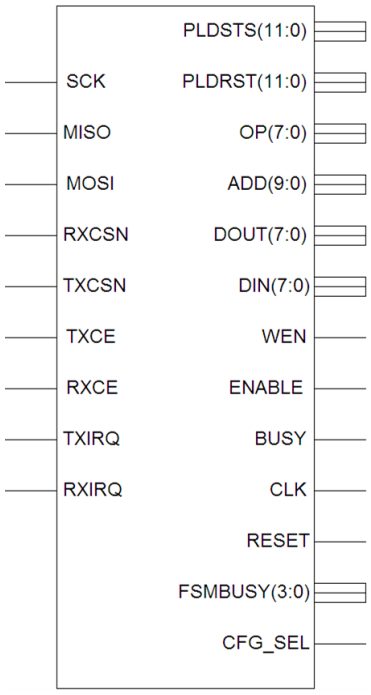


Figure 89 - RF Host Interface signals

3.3.4.1.2 Initialization

After a reset, the initialization is performed automatically when the ENABLE signal is driven to logic ‘1’.

During initialization, the BUSY signal is active (logic ‘1’) and the application must wait until completion before starting other operations.

3.3.4.1.3 Configuration

The nRF24L01 chip registers are shadowed internally to enable ease of access and speed. They can be read or written by the following operation modes (OP):

OP mode	Description
0x00	Normal Rx/Tx operation
0x02	Update registers shadow from all devices
0x03	Write registers shadow to all devices
0x04	Write register shadow to Rx device
0x05	Write register shadow to Tx device

Table 27 - Configuration operation modes

By loading the operation mode 0x00 to the OP bus, the RF Host resumes normal Rx/Tx operation.

3.3.4.1.4 Receiving data

Whenever data is received, the corresponding bitmap flag (PLDSTS) is updated, signaling the application that a new packet is ready for reception.

The application then reads the packet contents, after which it clears the bitmap flag by driving the corresponding bitmap reset flag (PLDRST) bit.

No further transmission on the datapipe is possible until the flag is cleared.

3.3.4.1.5 Transmitting data

In order to transmit data over a datapipe, the application must first check whether the datapipe is free for new data by checking the corresponding bitmap flag (PLDSTS), if so it can start writing the new packet data into the RF Host internal memory.

After this operation is completed, the data transmission is triggered by clearing the corresponding bitmap bit.

After successful transmission of the packet, the bitmap bit is set.

3.3.5 Wireless communications detailed schematic

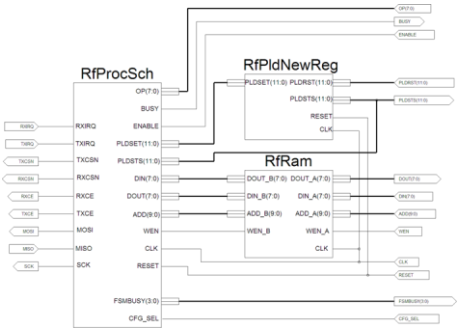


Figure 90 - RF Host

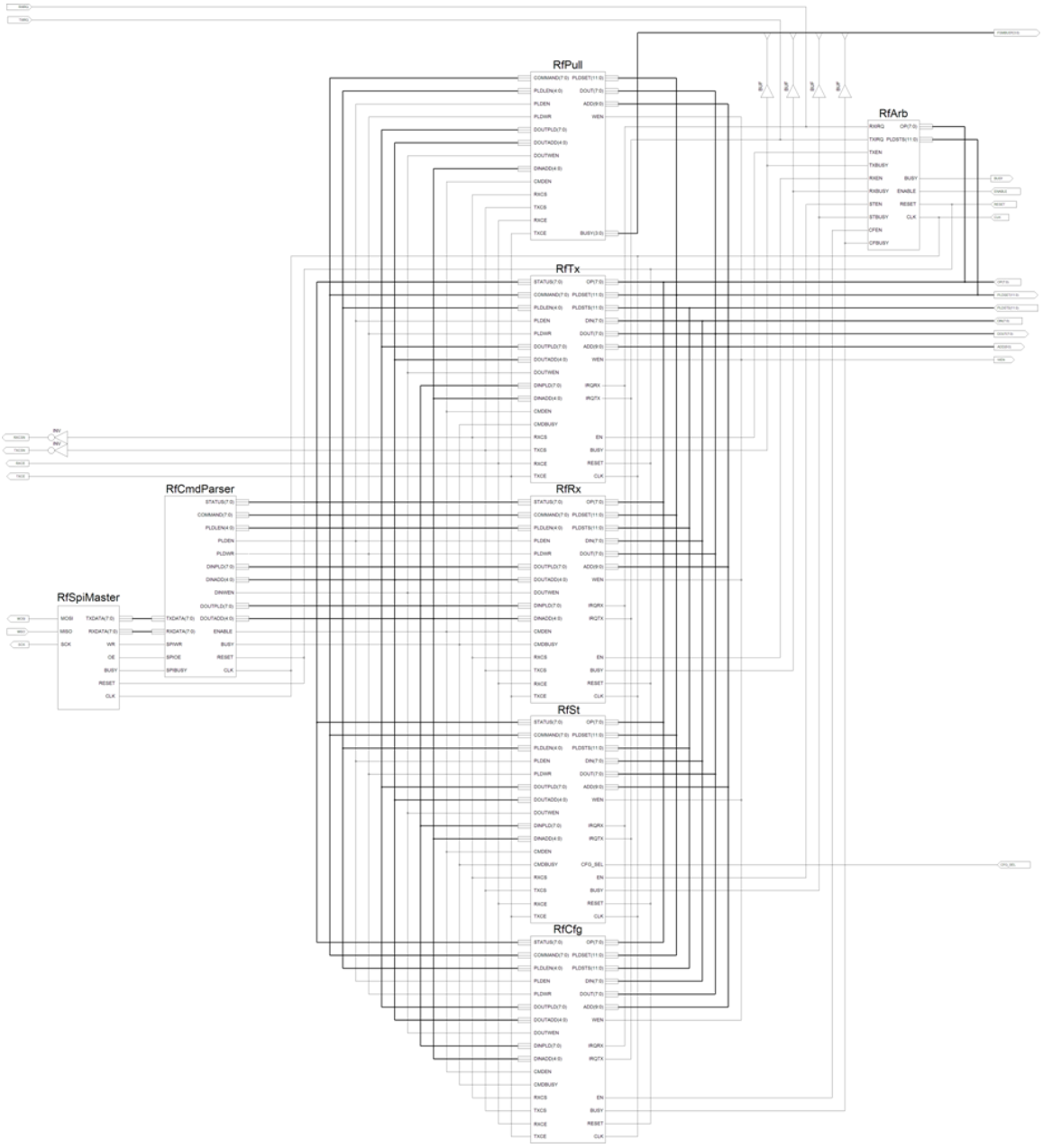


Figure 91 - RF processor

4 Results

4.1 Summary

In this section, all three main components will be analyzed in terms of performance and resource consumption. It is divided in the tree IP cores developed: MMC host, USB device and wireless communications.

4.2 MMC host

4.2.1 Performance

This host was synthesized with Xilinx's ISE tool for a Spartan 3E 1600 and is capable of operation up to 61.2MHz, more than enough for the maximum clock speeds defined at the JEDEC MMC standard (52 MHz for high speed access mode).

Data transfer performance however is not 100% optimized, as only single block transfers were implemented at this time. Nevertheless the preliminary results are satisfactory; the achieved transfer bandwidth for several MMC card brands can be seen in Table 28:

Brand	Capacity	Read Bandwidth	Write Bandwidth
Kingston	256 MByte	2.2 MBps	1.4 MBps
Extrememory	512 MByte	1,3 MBps	1,1 MBps
Transcend	512 MByte	1,1 MBps	730 kBps

All tests were made at 25MHz MMC bus clock speed, 1bit data interface.

Table 28 – Measured MMC host performance

As all blocks are read in single block transfers, it is true to say that the performance displayed on Table 28 is the random read performance, being the most conservative bandwidth test that is possible to perform in a flash card.

4.2.2 Resource consumption

The implemented design has a very small footprint, as it uses approximately 546 slices, and this is the main feature of this core.

Although its performance could be improved to accommodate multiple block read / write thus boosting performance up to MMC card's limit (up to 52 MBps theoretical maximum), that work is left for future development as the current design is more than enough for a single ECU cell.

Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	352	29,504	1%
Number of 4 input LUTs	754	29,504	2%
Logic Distribution			
Number of occupied Slices	546	14,752	3%
Number of Slices containing only related logic	546	546	100%
Number of Slices containing unrelated logic	0	546	0%
Total Number of 4 input LUTs	836	29,504	2%
Number used as logic	754		
Number used as a route-thru	18		
Number used for Dual Port RAMs	64		
Number of bonded IOBs	114	250	45%
Number of Block RAMs	2	36	5%
Number of GCLKs	1	24	4%
Total equivalent gate count for design	143,045		
Additional JTAG gate count for IOBs	5,472		

Table 29 - MMC host device utilization summary

4.3 USB device

4.3.1 Performance

4.3.1.1 Single device @ Root hub

This core is capable of operating with a clock speed of 95 MHz on a Spartan 3E speed grade -4. This is more than enough, given that the maximum allowable clock speed to the Slave FIFO interface is 60MHz in 8 bit mode or 30MHz in 16 bit mode.

A simple test is performed by transferring random data from / to the ECU Cell, without any kind of processing at the USB Host, to measure raw speeds. Results can be seen in Table 30:

Test	IN bandwidth	OUT bandwidth	Total bandwidth
IN	32 MB/s	N.A.	32 MB/s
OUT	N.A.	24 MB/s	24 MB/s
IN+OUT	24 MB/s	22 MB/s	46 MB/s

All tests were made at 48MHz Slave FIFO bus clock speed, 8bit data interface.

Table 30 – Measured USB Interface performance

4.3.1.2 Multiple devices @ Root hub

For multiple ECU Cell, the IN bandwidth is the most critical aspect to consider, as it will define the datalogging readout operation time, which should be as short as possible. A measurement, this time using the Client software to control transfers, was performed and the results are in Table 31:

Buffer (kB)	USB IN Performance @ HUB (MB/s)		
	1 Module	2 Modules	3 Modules
16	11	16	19
32	14	21	26
64	16	25	30
128	17	25	32

All tests were made at 48MHz Slave FIFO bus clock speed, 8bit data interface.

Table 31 - Multiple ECU Cell performance at root hub

4.3.1.3 Multiple devices @ Hub

This test gives a more realistic view of the expected performance of the complete system, as a USB Hub is used to connect three ECU Cells to the same root hub port. This test is similar to the previous one, results are in Table 32

Buffer (kB)	USB IN Performance @ HUB (MB/s)		
	1 Module	2 Modules	3 Modules
16	10	15	19
32	13	20	25
64	15	23	29
128	16	24	32

All tests were made at 48MHz Slave FIFO bus clock speed, 8bit data interface.

Table 32 - Multiple ECU Cell performance at local hub

4.3.1.4 Overall performance

From the measured tests it can be seen that a high performance solution was achieved. The bandwidth degradation due to the insertion of a hub in the bus is more pronounced with only one module, being the bandwidth loss lower than 1MBps (less than 6% over the maximum).

Also a fact to point out is that USB provides high bandwidth, with some degree of scalability, and one of the most important aspects to retain is that the addition of devices does not impair overall performance, but in instance shares available bandwidth more efficiently among devices.

Also in the scalability aspect of the USB, as long as there is available bandwidth to accommodate more transfers. It can be seen that with 128kB buffers at the Host, the performance for one module is 16MBps, two modules is 24MBps (+50%) and for three modules the total bandwidth is 32MBps (+100%).

4.3.2 USB Interface resource consumption

Relying in simple design to achieve high performance generally yields low resource consumption, as is the case with this core.

Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	474	29,504	1%
Number of 4 input LUTs	533	29,504	1%
Logic Distribution			
Number of occupied Slices	453	14,752	3%
Number of Slices containing only related logic	453	453	100%
Number of Slices containing unrelated logic	0	453	0%
Total Number of 4 input LUTs	720	29,504	2%
Number used as logic	533		
Number used as a route-thru	187		
Number of bonded IOBs	114	250	45%
Number of Block RAMs	4	36	11%
Number of GCLKs	2	24	8%
Number of DCMs	1	8	12%
Total equivalent gate count for design	277,922		
Additional JTAG gate count for IOBs	5,472		

Table 33 - USB Interface device utilization summary

The core footprint is reduced, with a consumption of only 453 slices and 4 Block RAM. This is accomplished because all of the complex USB physical and logical layers are already implemented on the CY7C68013, allowing the FPGA to allocate resources to other applications, such as a more complex ECU Cell core.

4.4 Wireless communications

4.4.1 Performance

This core can be clocked up to 83 MHz on a Spartan 3E 1600 -4. This is more than enough for the internal logic, as the SPI clock cannot be faster than 5MHz.

Using the full Enhanced Shock-Burst capabilities of the nRF24L01 (auto acknowledge and retransmission, crc generation and validation, preamble and device addressing) and without extensive use of the buffering capabilities of the nRF24L01, about 30kBps were achieved. This is roughly 38% of the maximum capacity available, which is highly dependent on the packet size and the bit error rate.

Higher throughputs are possible, up to 60kBps @ 0,1% BER, and given the optimum transmission conditions, up to 80kBps is possible if BER< 0,007%.

But for example, if auto-acknowledge and retransmission were not to be implemented on this design, the effective throughput could be increased, and speeds up to 100kBps could be achieved given that BER < 0,033%.

4.4.2 Resource consumption

The RF Host core resource consumption is as follows:

Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	189	29,504	1%
Number of 4 input LUTs	516	29,504	1%
Logic Distribution			
Number of occupied Slices	311	14,752	2%
Number of Slices containing only related logic	311	311	100%
Number of Slices containing unrelated logic	0	311	0%
Total Number of 4 input LUTs	572	29,504	1%
Number used as logic	516		
Number used for Dual Port RAMs	48		

Number used as 16x1 RAMs	8		
Number of bonded IOBs	114	250	45%
IOB Flip Flops	3		
Number of Block RAMs	1	36	2%
Number of GCLKs	1	24	4%
Total equivalent gate count for design	74,507		
Additional JTAG gate count for IOBs	5,472		

Table 34 - Wireless communications resource consumption

This core uses only 311 slices and 1 Block RAM, which makes this core small yet the overall design is efficient and highly configurable, ideal to use in the ECU Cell core.

Also, this small footprint enables the instantiation of multiple cores to take advantage of the low price of the nRF24L01 modules and available scalability potential by efficiently usage of the 41 available 2Mbit channels in order to achieve higher throughputs.

5 Conclusions

5.1 Summary

This chapter compares the obtained results with the proposed objectives, focusing in each one of the three IP cores developed.

5.2 MMC Host

The proposed design is able to cope with the objectives requirements. Each module easily supports up to 2GB cards, the maximum size allowed by the standard, allowing 58 hours of historical data recording with the 35MB/(hour × 1000variables/sec) assumption.

Also download capability is high, having reached 2,2MB/sec in performed tests. In this case there's the possibility to increase performance, implementing block read support at the MMC host and/or increasing the data bus width to 4 or even 8 bits, although it should be only considered if developing the support for "MMC Electrical Standard, High Capacity (MMCA 4.2)" that would extend maximum capacity to 32GB.

5.3 USB peripheral

The developed solution is capable to deliver high bandwidth, largely surpassing the objectives. The proposed design is able to deliver a bandwidth of 32MB/sec spanned across 3 modules connected to the same hub, yielding 10,7MB/sec per module. This test demonstrates that it is possible to achieve a high bandwidth solution, even though the USB bus is shared by several devices.

One other important aspect of the developed solution is that, device operation as a USB peripheral makes installation easy in a new computer, and seamless in a computer that had the device drivers pre-installed.

5.4 Wireless communications

One of the most emphasized objectives was scalability, and the option to use a proprietary protocol stack, the Nordic nRF24L01, proves to be the best among the solutions analyzed. It is able to provide the minimum bandwidth required, and to divide the available spectrum in 42 non overlapping 2Mbps channels, allowing easy accommodation of many devices in close range, given that a frequency hopping or similar algorithm is deployed (14) (19).

Although the nRF24L01 uses a proprietary communication stack, it is reduced both in complexity and in scope, as it only regards the physical and link layer. Also, most of the link layer functionalities can be disabled, circumventing its embedded algorithm, allowing the deployment of different, but emulated, protocols.

Further enhancements could be made by using a Bluetooth similar algorithm, in which all the devices connected in a network that share the same frequency channel pre-negotiate a timeslot assignment in order to reduce collisions, allowing easy accommodation of two transmitters per channel.

If further performance is needed it is possible to include multiple instances of the RF module, each with its processor, or to modify the RF block in order to accommodate more RF interfaces. This is possible due to the small footprint and high performance of the developed IP Core. As an example, if 10 chip pairs were to be used, a bandwidth of 10Mbps in full duplex could be easily achieved, while maintaining some degree scalability.

Glossary

ALOHAnet ---------- Computer networking system developed at the University of Hawaii

ASIC ---------- Application Specific Integrated Circuit

ATA ---------- Advanced Technology Attachment

BER ---------- Bit Error Rate

BOM ---------- Bill of Materials

CRC ---------- Cyclic Redundancy Check

CWDM ---------- Coarse Wavelength Division Multiplexing

C# ---------- C Sharp

DPSK ---------- Differential Phase Shift Keying

ECC ---------- Error Correction Codes

ECU ---------- Electronic Control Unit

EDC ---------- Error Detection and Correction

EISA ---------- Extended Industry Standard Architecture

FHSS ---------- Frequency Hopping Spread Spectrum

FIFO ---------- First In First Out

FPGA ---------- Field Programmable Gate Array

FSM ---------- Finite State Machine

GFSK ---------- Gaussian Filtered Shift Keying

GPIF ---------- General Purpose Interface

IDMS ---------- Integrated Development & Management System

IEEE ---------- Institute of Electrical and Electronic Engineers

IIC ---------- Inter-Integrated Circuit

IP ---------- Internet Protocol

IP cores ---------- Semiconductor intellectual property core

ISM ---------- Industrial, Scientific and Medical

MIMO ---------- Multiple Input Multiple Output

MMC ---------- Multimedia Card

MSB ---------- Most Significant Byte

PCI---------- Peripheral Component Interconnect

PCMCIA ---------- Personal Computer Memory Card International Association

PSRR ---------- Power Supply Rejection Ratio

RAM ---------- Random Access Memory

RS232---------- Serial single ended data and control signals standard

RTL ---------- Register Transfer Language

SDA ---------- Secure Digital Association

SoC---------- System on Chip

SPI---------- Serial Peripheral Interface

SPP ---------- Serial Port Profile

SSOP ---------- Shrink Small-Outline Package

USART---------- Universal Asynchronous Receiver/Transmitter

USB ---------- Universal Serial Bus

Utopia---------- System Packet Interface ATM specification

VHDL---------- VHSIC Hardware Description Language

WLAN ---------- Wireless Local Area Network

Bibliography

1. **Anderson, Don.** *Firewire System Architecture, Second Edition.* 1998.

2. **Microsoft Corporation.** IP networking over the IEEE 1394 bus is not supported in Windows Vista and in all later versions of Windows. *Microsoft Support.* [Online] Microsoft , November 28, 2007. [Cited: September 4, 2010.] http://support.microsoft.com/kb/943719.

3. **USB Implementers Forum.** *Universal Serial Bus Specification Revision 2.0.* s.l. : Compaq Computer Corporation, Hewlett-Packard Company, Intel Corporation, Lucent Technologies Inc, Microsoft Corporation, NEC Corporation, Koninklijke Philips Electronics N.V., 2000.

4. **PCWorld.** SuperSpeed USB 3.0: More Details Emerge. *PCWorld.* [Online] PCWorld, January 7, 2009. [Cited: Spetember 4, 2010.] http://www.pcworld.com/article/156494/superspeed_usb_30_more_details_emerge.html.

5. **Smithsonian.** National Museum of American History. [Online] 1992. [Cited: September 3, 2010.] http://americanhistory.si.edu/collections/object.cfm?key=35&objkey=96.

6. **Pearson, Jamie Parker.** *Digital at Work.* s.l. : Digital Press, 1992.

7. **Nanda, Amit and Nalder, Greg.** *AN14557: Developing USB Applications.* s.l. : Cypress Semiconductor Corporation, 2007.

8. **wndw.net.** *Wireless Networking in the Developing World.* 2007.

9. **IEEE.** *802.15.1-2005.* 2005.

10. **ZigBee Alliance.** *ZigBee Specification.* 2008.

11. **Nordic Semiconductor ASA.** *Low Cost Networks, ZigBee™ & 802.15.4.* Tiller, Norway : Nordic Semiconductor ASA, 2005.

12. —. *nRF24L01: Single Chip 2.4GHz Transceiver Product Specification.* Tiller, Norway : Nordic Semiconductor ASA, 2007.

13. —. *nAN24-07: Frequency Agility Protocol for nRF24xx.* Tiller, Norway : Nordic Semiconductor ASA, 2004.

14. **Popovski, Petar, Yomo, Hiroyuki and Prasad, Ramjee.** *Strategies for Adaptative Frequency Hopping in the Unlicenced Bands.* s.l. : IEEE, 2006.

15. **MultiMedia Card Alliance.** *JESD84-B41.* s.l. : JEDEC SOLID STATE TECHNOLOGY ASSOCIATION, 2007.

16. **Digilent.** *Digilen PmodUSB2 Module Board Reference Manual.* s.l. : Digilent, Inc, 2005.

17. **Cypress Semiconductor Corporation.** *38-08032 Rev.L: EZ-USB FX2LP™ USB Microcontroller: High-Speed USB Peripheral Controller.* San Jose, CA : Cypress Semiconductor Corporation, 2008.

18. **Nordic Semiconductor ASA.** *nRF24L01-REFMOD: nRF24L01 Reference Modules.* Tiller, Norway : Nordic Semiconductor ASA, 2006.

19. —. *RF communication in a multi-user environment.* Tiller, Norway : Nordic Semiconductor ASA, 2003.

20. **Intel.** *High Speed USB Platform Design Guidelines Rev. 1.0.* s.l. : Intel Corporation, 2001.

21. **Cypress Semiconductor Corporation.** *001-13670 Rev.A: EZ-USB® Technical Reference Manual.* San Jose, CA : Cypress Semiconductor Corporation, 2007.

Table of contents

Table of figures

1 USB Interface detailed schematic

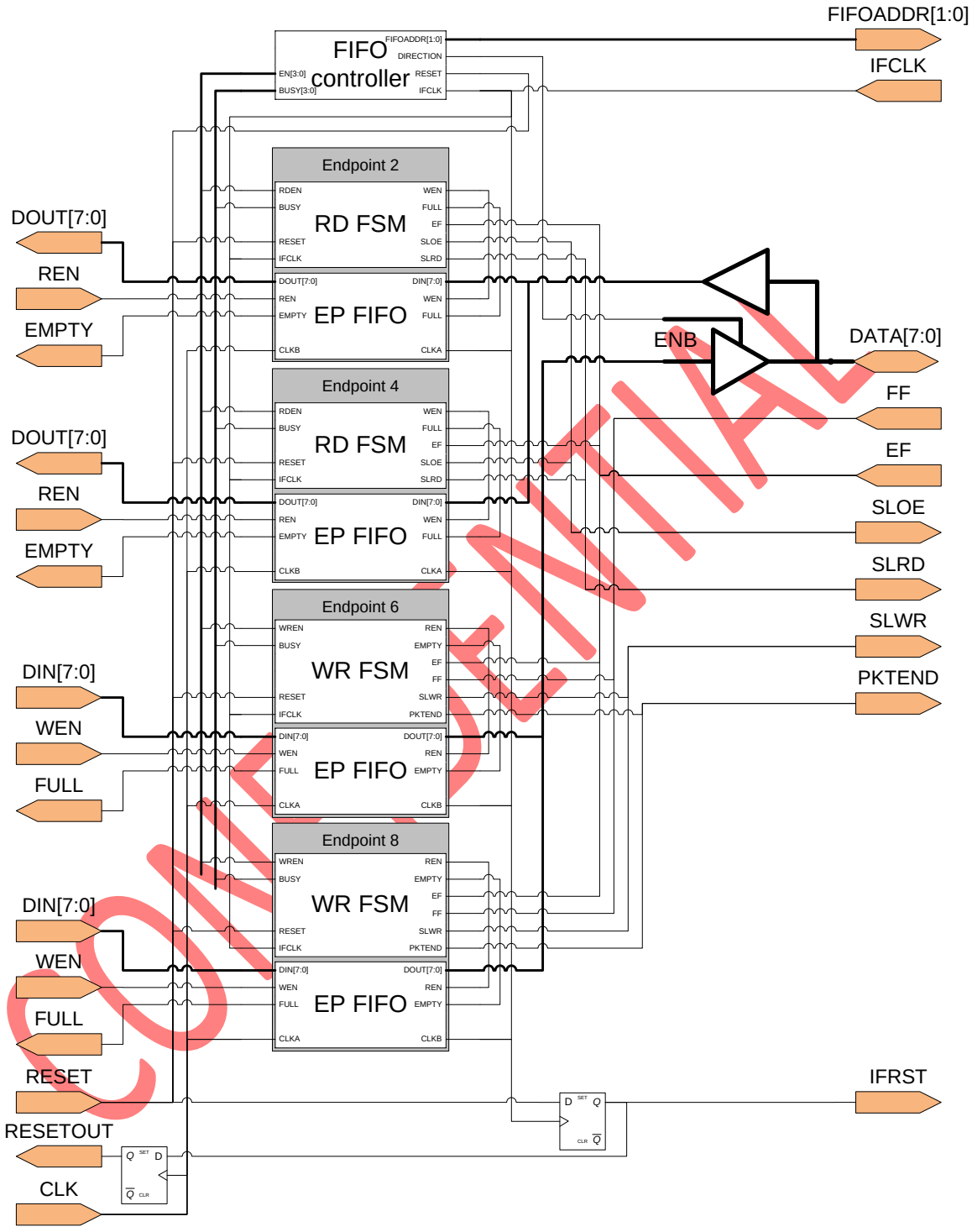


Figure 1 - USB Interface detailed schematic

1.1 USB Host

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity USBInterface is
   port ( DATAINOUT: inout std_logic_vector (7 downto 0);    -- Cypress data inout
          EF       : in    std_logic;                              -- Cypress empty flag
          FF       : in    std_logic;                              -- Cypress full flag
          FIFOADDR : out   std_logic_vector (1 downto 0);    -- Cypress fifo address
          PKTEND   : out   std_logic;                              -- Cypress packet end
          SLOE     : out   std_logic;                              -- Cypress output
enable
          SLRD     : out   std_logic;                              -- Cypress read
          SLWR     : out   std_logic;                              -- Cypress write
          IFCLK    : in    std_logic;                              -- Cypress clock
             IFRST   : out  std_logic;                              -- Cypress reset output
          EP2REN   : in    std_logic;                              -- Processor EP2 read
enable
          EP4REN   : in    std_logic;                              -- Processor EP4 read
enable
          EP6WEN   : in    std_logic;                              -- Processor EP6 write
enable
          EP8WEN   : in    std_logic;                              -- Processor EP8 write
enable
          EP2DOUT  : out   std_logic_vector (7 downto 0);    -- Processor EP2 data out
          EP4DOUT  : out   std_logic_vector (7 downto 0);    -- Processor EP4 data out
          EP6DIN   : in    std_logic_vector (7 downto 0);    -- Processor EP6 data in
          EP8DIN   : in    std_logic_vector (7 downto 0);    -- Processor EP8 data in
          EP2AEMPTY: out   std_logic;                              -- Processor EP2 almost
empty
          EP2EMPTY : out   std_logic;                              -- Processor EP2 empty
          EP4AEMPTY: out   std_logic;                              -- Processor EP4 almost
empty
          EP4EMPTY : out   std_logic;                              -- Processor EP4 empty
          EP6AFULL : out   std_logic;                              -- Processor EP6 almost
full
          EP6FULL  : out   std_logic;                              -- Processor EP6 full
          EP8AFULL : out   std_logic;                              -- Processor EP8 almost
full
          EP8FULL  : out   std_logic;                              -- Processor EP8 full
          MCLK     : in    std_logic;                              -- Processor clock
          RESET    : in    std_logic;                              -- Processor reset
input
             RESETOUT : out std_logic                              -- Processor reset
output
          );
end USBInterface;

architecture BEHAVIORAL of USBInterface is
   attribute BOX_TYPE   : string ;
   signal BUSY          : std_logic_vector (3 downto 0);
   signal DATA0         : std_logic_vector (7 downto 0);
   signal DATA1         : std_logic_vector (7 downto 0);
   signal DATA2         : std_logic_vector (7 downto 0);
   signal DATA3         : std_logic_vector (7 downto 0);
   signal EN            : std_logic_vector (3 downto 0);
   signal EP2FULL       : std_logic;
   signal EP2WEN        : std_logic;
   signal EP4FULL       : std_logic;
   signal EP4WEN        : std_logic;
   signal EP6EMPTY      : std_logic;
```

```
signal EP6REN        : std_logic;
signal EP8EMPTY      : std_logic;
signal EP8REN        : std_logic;
signal PKTEND2       : std_logic;
signal PKTEND3       : std_logic;
signal SLOE0         : std_logic;
signal SLOE1         : std_logic;
signal SLRD0         : std_logic;
signal SLRD1         : std_logic;
signal SLWR2         : std_logic;
signal SLWR3         : std_logic;
 signal s_reset          : std_logic;
 signal cyclk            : std_logic;
 signal s_cyDcmLocked    : std_logic;

 signal data_in          : std_logic_vector(7 downto 0);
 signal data_out     : std_logic_vector(7 downto 0);
 signal s_dir            : std_logic;

 COMPONENT CyDCM
 PORT(
     CLKIN_IN : IN std_logic;
     RST_IN : IN std_logic;
     CLKIN_IBUFG_OUT : OUT std_logic;
     CLK0_OUT : OUT std_logic;
     LOCKED_OUT : OUT std_logic
     );
 END COMPONENT;

component CyMaster
   port ( RESET     : in    std_logic;
          BUSY      : in    std_logic_vector (3 downto 0);
          EN        : out   std_logic_vector (3 downto 0);
          DIRECTION : out   std_logic;
          FIFOADDR  : out   std_logic_vector (1 downto 0);
          IFCLK     : in    std_logic);
end component;

component CyMux
   port ( SLOE1    : in    std_logic;
          SLRD1    : in    std_logic;
          SLWR1    : in    std_logic;
          PKTEND1  : in    std_logic;
          SLRD0    : in    std_logic;
          SLWR0    : in    std_logic;
          PKTEND0  : in    std_logic;
          SLOE0    : in    std_logic;
          DATA0    : out   std_logic_vector (7 downto 0);
          SLOE2    : in    std_logic;
          SLRD2    : in    std_logic;
          SLWR2    : in    std_logic;
          PKTEND2  : in    std_logic;
          DATA3    : in    std_logic_vector (7 downto 0);
          DATA1    : out   std_logic_vector (7 downto 0);
          SLOE3    : in    std_logic;
          SLRD3    : in    std_logic;
          SLWR3    : in    std_logic;
          PKTEND3  : in    std_logic;
          DATA2    : in    std_logic_vector (7 downto 0);
          EN       : in    std_logic_vector (3 downto 0);
          SLOE     : out   std_logic;
          SLRD     : out   std_logic;
          SLWR     : out   std_logic;
          PKTEND   : out   std_logic;
```

```
            DATAIN  : in    std_logic_vector (7 downto 0);
            DATAOUT : out   std_logic_vector (7 downto 0));
    end component;

    component CyRead
       port ( RESET : in    std_logic;
              WEN   : out   std_logic;
              FULL  : in    std_logic;
              RDEN  : in    std_logic;
              BUSY  : out   std_logic;
              SLRD  : out   std_logic;
              SLOE  : out   std_logic;
              EF    : in    std_logic;
              IFCLK : in    std_logic);
    end component;

     component CyFifo
         port ( din: IN std_logic_VECTOR(7 downto 0);
                  rd_clk: IN std_logic;
                  rd_en: IN std_logic;
                  rst: IN std_logic;
                  wr_clk: IN std_logic;
                  wr_en: IN std_logic;
                  almost_empty: OUT std_logic;
                  almost_full: OUT std_logic;
                  dout: OUT std_logic_VECTOR(7 downto 0);
                  empty: OUT std_logic;
                  full: OUT std_logic);
     end component;

    component CyWrite
       port ( EMPTY  : in    std_logic;
              REN    : out   std_logic;
              IFCLK  : in    std_logic;
              WREN   : in    std_logic;
              RESET  : in    std_logic;
              BUSY   : out   std_logic;
              PKTEND : out   std_logic;
              SLWR   : out   std_logic;
              EF     : in    std_logic;
              FF     : in    std_logic);
    end component;

     signal sInternalReset    : std_logic;
     signal sResetCounter        : std_logic_vector(17 downto 0) := (others=>'1');
    signal sDcmReset        : std_logic := '1';
    signal sResetByDCM      : std_logic := '1';

    type SM_States is (S_POR,
                       S_WAITDCMLOCK,
                       S_IDLE,
                       S_RESETDCM);
     signal state            : SM_States := S_POR;
begin

     process(MCLK, RESET) begin
         if((RESET='1') or ((sDcmReset='1') and (state=S_WAITDCMLOCK)))then
             sResetCounter   <= (others=>'1');
         elsif(rising_edge(MCLK))then
             if(sResetCounter>conv_std_logic_vector(0,18))then
                 sResetCounter   <= sResetCounter - '1';
             end if;
         end if;
     end process;
```

```
process(MCLK, RESET) begin
   if(RESET='1')then
      sDcmReset   <= '1';
      sResetByDCM <= '1';
      state       <= S_POR;
   elsif(rising_edge(MCLK)) then
         case state is
             when S_POR =>
            sInternalReset <= '1';
            sResetByDCM    <= '1';
            if((sDcmReset='1') and (sResetCounter=conv_std_logic_vector(0,18))) then
               state       <= S_WAITDCMLOCK;
            else
               sDcmReset   <= '1';
            end if;
         when S_WAITDCMLOCK =>
            sInternalReset <= '0';
            sResetByDCM    <= '1';
            sDcmReset      <= '0';
            if(s_cyDcmLocked='1') then
               state <= S_IDLE;
            elsif((sResetCounter=conv_std_logic_vector(0,18)) and (sDcmReset='0'))then
               state <= S_RESETDCM;
            end if;
         when S_IDLE =>
            sInternalReset <= '0';
            sResetByDCM    <= '0';
            if(s_cyDcmLocked='0') then
               sResetByDCM <= '1';
               state       <= S_RESETDCM;
            end if;
         when S_RESETDCM =>
            sInternalReset <= '0';
            sResetByDCM    <= '1';
            if(sDcmReset='1') then
               state       <= S_WAITDCMLOCK;
            else
               sDcmReset   <= '1';
            end if;
         when others =>
             sDcmReset      <= '1';
             sInternalReset <= '1';
             sResetByDCM    <= '1';
             state          <= S_POR;
      end case;
   end if;
end process;


   IFRST       <= not sInternalReset;
   s_reset  <= sInternalReset or sResetByDCM;
   RESETOUT    <= sInternalReset;
   data_in <= DATAINOUT;
   DATAINOUT <= data_out when (s_dir='0') else (others=>'Z');

   UCLK: CyDCM PORT MAP(
       CLKIN_IN => IFCLK,
       RST_IN => sDcmReset,
--     CLKIN_IBUFG_OUT => ,
       CLK0_OUT => cyclk,
       LOCKED_OUT => s_cyDcmLocked
   );
```

```
    EPMASTER : CyMaster
       port map (BUSY(3 downto 0)=>BUSY(3 downto 0),
                 IFCLK=>cyclk,
                 RESET=>s_reset,
                 DIRECTION=>s_dir,
                 EN(3 downto 0)=>EN(3 downto 0),
                 FIFOADDR(1 downto 0)=>FIFOADDR(1 downto 0));

    EPMUX : CyMux
       port map (DATAIN(7 downto 0)=>data_in(7 downto 0),
                 DATA2(7 downto 0)=>DATA2(7 downto 0),
                 DATA3(7 downto 0)=>DATA3(7 downto 0),
                 EN(3 downto 0)=>BUSY(3 downto 0),
                 PKTEND0=>'1',
                 PKTEND1=>'1',
                 PKTEND2=>PKTEND2,
                 PKTEND3=>PKTEND3,
                 SLOE0=>SLOE0,
                 SLOE1=>SLOE1,
                 SLOE2=>'1',
                 SLOE3=>'1',
                 SLRD0=>SLRD0,
                 SLRD1=>SLRD1,
                 SLRD2=>'1',
                 SLRD3=>'1',
                 SLWR0=>'1',
                 SLWR1=>'1',
                 SLWR2=>SLWR2,
                 SLWR3=>SLWR3,
                 DATAOUT(7 downto 0)=>data_out(7 downto 0),
                 DATA0(7 downto 0)=>DATA0(7 downto 0),
                 DATA1(7 downto 0)=>DATA1(7 downto 0),
                 PKTEND=>PKTEND,
                 SLOE=>SLOE,
                 SLRD=>SLRD,
                 SLWR=>SLWR);

    EP2FSM : CyRead
       port map (EF=>EF,
                 FULL=>EP2FULL,
                 IFCLK=>cyclk,
                 RDEN=>EN(0),
                 RESET=>s_reset,
                 BUSY=>BUSY(0),
                 SLOE=>SLOE0,
                 SLRD=>SLRD0,
                 WEN=>EP2WEN);

     EP2MEM : CyFifo
        port map (din => DATA0,
                  rd_clk => MCLK,
                  rd_en => EP2REN,
                  rst => s_reset,
                  wr_clk => cyclk,
                  wr_en => EP2WEN,
                  almost_empty => EP2AEMPTY,
--                  almost_full => ,
                  dout => EP2DOUT,
                  empty => EP2EMPTY,
                  full => EP2FULL);

    EP4FSM : CyRead
       port map (EF=>EF,
                 FULL=>EP4FULL,
```

```
                IFCLK=>cyclk,
                RDEN=>EN(1),
                RESET=>s_reset,
                BUSY=>BUSY(1),
                SLOE=>SLOE1,
                SLRD=>SLRD1,
                WEN=>EP4WEN);

    EP4MEM : CyFifo
        port map (din => DATA1,
                  rd_clk => MCLK,
                  rd_en => EP4REN,
                  rst => s_reset,
                  wr_clk => cyclk,
                  wr_en => EP4WEN,
                  almost_empty => EP4AEMPTY,
--                almost_full => ,
                  dout => EP4DOUT,
                  empty => EP4EMPTY,
                  full => EP4FULL);

   EP6FSM : CyWrite
      port map (EF=>EF,
                EMPTY=>EP6EMPTY,
                FF=>FF,
                IFCLK=>cyclk,
                RESET=>s_reset,
                WREN=>EN(2),
                BUSY=>BUSY(2),
                PKTEND=>PKTEND2,
                REN=>EP6REN,
                SLWR=>SLWR2);

    EP6MEM : CyFifo
        port map (din => EP6DIN,
                  rd_clk => cyclk,
                  rd_en => EP6REN,
                  rst => s_reset,
                  wr_clk => MCLK,
                  wr_en => EP6WEN,
--                almost_empty => ,
                  almost_full => EP6AFULL,
                  dout => DATA2,
                  empty => EP6EMPTY,
                  full => EP6FULL);

   EP8FSM : CyWrite
      port map (EF=>EF,
                EMPTY=>EP8EMPTY,
                FF=>FF,
                IFCLK=>cyclk,
                RESET=>s_reset,
                WREN=>EN(3),
                BUSY=>BUSY(3),
                PKTEND=>PKTEND3,
                REN=>EP8REN,
                SLWR=>SLWR3);

    EP8MEM : CyFifo
        port map (din => EP8DIN,
                  rd_clk => cyclk,
                  rd_en => EP8REN,
                  rst => s_reset,
                  wr_clk => MCLK,
```

```
                    wr_en => EP8WEN,
--                  almost_empty => ,
                    almost_full => EP8AFULL,
                    dout => DATA3,
                    empty => EP8EMPTY,
                    full => EP8FULL);

end BEHAVIORAL;
```

1.2 Bus arbitrer

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CyMaster is
    Port ( IFCLK        : in  STD_LOGIC;
           DIRECTION    : out  STD_LOGIC;
           FIFOADDR : out  STD_LOGIC_VECTOR (1 downto 0);
           EN           : out  STD_LOGIC_VECTOR (3 downto 0);
           BUSY     : in  STD_LOGIC_VECTOR (3 downto 0);
              RESET     : in STD_LOGIC);
end CyMaster;

architecture CyMaster_rtl of CyMaster is
    constant MAX_IN_COUNT   : integer := 65535;
    constant MAX_OUT_COUNT  : integer := 65535;
    constant MAX_WAIT_COUNT : integer := 3;
    type SM_States is (S_Wait, S_Select, S_Abort);
    signal state            : SM_States := S_Select;
    signal selected_ep  : STD_LOGIC_VECTOR(1 downto 0);
    signal out_counter  : integer range 0 to MAX_OUT_COUNT := 0;
    signal in_counter       : integer range 0 to MAX_IN_COUNT := 0;
    signal wait_counter : integer range 0 to MAX_WAIT_COUNT := 0;
    signal out_mode     : STD_LOGIC;
    signal current_BUSY : STD_LOGIC;
    signal current_EN       : STD_LOGIC_VECTOR(3 downto 0);
    signal enable_EN        : STD_LOGIC;
begin
    FIFOADDR <= selected_ep;
    DIRECTION <= out_mode;
    out_mode <= '0' when ((selected_ep = "10") or (selected_ep = "11")) else '1';
    current_BUSY <= BUSY(0) when (selected_ep="00") else
                    BUSY(1) when (selected_ep="01") else
                    BUSY(2) when (selected_ep="10") else
                    BUSY(3) when (selected_ep="11") else '0';
    EN <= current_EN when (enable_EN='1') else (others=>'0');
    current_EN <= "0001" when (selected_ep = "00") else
                  "0010" when (selected_ep = "01") else
                  "0100" when (selected_ep = "10") else
                  "1000" when (selected_ep = "11") else "0000";

    process(IFCLK, RESET) begin
      if(RESET='1') then
          in_counter <= 0;
          out_counter <= 0;
          enable_EN <= '0';
       elsif(rising_edge(IFCLK)) then
          case state is
              when S_Wait =>
                  enable_EN       <= '1';
                  if(current_BUSY = '1') then
                      if(out_mode = '1') then
                          if (in_counter >= MAX_IN_COUNT) then
                              state           <= S_Abort;
                          else
                              in_counter  <= in_counter + 1;
                          end if;
                      else
                          if (out_counter >= MAX_OUT_COUNT) then
                              state           <= S_Abort;
                          else
```

```
                        out_counter <= OUT_counter + 1;
                    end if;
                end if;
            else
                enable_EN   <= '0';
                state           <= S_Abort;
            end if;
        when S_Select =>
            in_counter  <= 0;
            out_counter <= 0;
            enable_EN   <= '1';
            if(current_BUSY='1' and enable_EN='1') then
                state       <= S_Wait;
            elsif(wait_counter >= MAX_WAIT_COUNT) then
                selected_ep     <= selected_ep + '1';
                wait_counter    <= 0;
            else
                wait_counter    <= wait_counter + 1;
            end if;
        when S_Abort =>
            if(current_BUSY='0') then
                selected_ep     <= selected_ep + '1';
                enable_EN       <= '1';
                wait_counter    <= 0;
                state               <= S_Select;
            else
                in_counter      <= 0;
                out_counter     <= 0;
                enable_EN       <= '0';
            end if;
        when others =>
            state               <= S_Select;
            wait_counter    <= 0;
            enable_EN       <= '0';
    end case;
  end if;
 end process;
end CyMaster_rtl;
```

1.3 BUS Mux

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity CyMux is
    Port ( SLOE :       out STD_LOGIC;
           SLRD :       out STD_LOGIC;
           SLWR :       out STD_LOGIC;
           PKTEND : out STD_LOGIC;
              DATAIN :  in STD_LOGIC_VECTOR(7 downto 0);
              DATAOUT : out STD_LOGIC_VECTOR(7 downto 0);
              SLOE0 :   in STD_LOGIC;
           SLRD0 :  in STD_LOGIC;
           SLWR0 :  in STD_LOGIC;
           PKTEND0 :    in STD_LOGIC;
              DATA0 :   out STD_LOGIC_VECTOR(7 downto 0);
              SLOE1 :   in STD_LOGIC;
           SLRD1 :  in STD_LOGIC;
           SLWR1 :  in STD_LOGIC;
```

```
            PKTEND1 :    in STD_LOGIC;
               DATA1 :   out STD_LOGIC_VECTOR(7 downto 0);
               SLOE2 :   in STD_LOGIC;
            SLRD2 :  in STD_LOGIC;
            SLWR2 :  in STD_LOGIC;
            PKTEND2 :    in STD_LOGIC;
               DATA2 :   in STD_LOGIC_VECTOR(7 downto 0);
               SLOE3 :   in STD_LOGIC;
            SLRD3 :  in STD_LOGIC;
            SLWR3 :  in STD_LOGIC;
            PKTEND3 :    in STD_LOGIC;
               DATA3 :   in STD_LOGIC_VECTOR(7 downto 0);
               EN :      in STD_LOGIC_VECTOR(3 downto 0)
               );
end CyMux;

architecture CyMux_rtl of CyMux is

    signal data_in :    STD_LOGIC_VECTOR(7 downto 0);
    signal data_out :   STD_LOGIC_VECTOR(7 downto 0);

begin
    SLOE <= SLOE0 when (EN="0001") else
              SLOE1 when (EN="0010") else
              SLOE2 when (EN="0100") else
              SLOE3 when (EN="1000") else '1';

    SLRD <= SLRD0 when (EN="0001") else
              SLRD1 when (EN="0010") else
              SLRD2 when (EN="0100") else
              SLRD3 when (EN="1000") else '1';

    SLWR <= SLWR0 when (EN="0001") else
              SLWR1 when (EN="0010") else
              SLWR2 when (EN="0100") else
              SLWR3 when (EN="1000") else '1';

    PKTEND <= PKTEND0 when (EN="0001") else
                 PKTEND1 when (EN="0010") else
                 PKTEND2 when (EN="0100") else
                 PKTEND3 when (EN="1000") else '1';

    DATA0 <= data_in;
    DATA1 <= data_in;

    data_out <= DATA2 when (EN="0100") else
                   DATA3 when (EN="1000") else (others=>'0');

    data_in <= DATAIN;
    DATAOUT <= data_out;

end CyMux_rtl;
```

1.4 Bus read

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CyRead is
    Port ( IFCLK    : in STD_LOGIC;
              RDEN  : in  STD_LOGIC;
           BUSY : out  STD_LOGIC;
           WEN      : out  STD_LOGIC;
           FULL : in  STD_LOGIC;
           EF       : in  STD_LOGIC;
           SLOE : out  STD_LOGIC;
           SLRD : out  STD_LOGIC;
              RESET : in STD_LOGIC
              );
end CyRead;

architecture CyRead_rtl of CyRead is
    type SM_States is (S_IDLE, S_SETUP, S_READ, S_EXIT, S_STALL);

    signal state            : SM_States := S_IDLE;
    signal sloe_a           : std_logic := '1';
    signal slrd_a           : std_logic := '1';
    signal wen_a            : std_logic := '0';
    signal slrd_en1     : std_logic;
    signal wen_en1          : std_logic;
    signal busy_a           : std_logic := '0';
begin

    BUSY <= busy_a;
    SLOE <= sloe_a;
    SLRD <= slrd_a or not slrd_en1;
    WEN <= wen_a and wen_en1;

    slrd_en1 <= (not FULL) and EF and busy_a;
    wen_en1 <= (not FULL) and EF and busy_a;

    process(IFCLK, RESET) begin
        if(rising_edge(IFCLK)) then
         if(RESET='1') then
           if(wen_a='0') then
              state <= S_IDLE;
              busy_a    <= '0';
           end if;
           if((sloe_a='1') and (slrd_a='1'))then
              wen_a         <= '0';
           end if;
           sloe_a  <= '1';
           slrd_a  <= '1';
        else
           case state is
              when S_IDLE =>
                 if(RDEN='1' and FULL='0' and EF='1') then
                    busy_a <= '1';
                    wen_a  <= '0';
                    sloe_a <= '1';
                    slrd_a <= '1';
                    state       <= S_SETUP;
                 else
                    busy_a <= '0';
                    wen_a       <= '0';
```

```
                sloe_a <= '1';
                slrd_a <= '1';
            end if;
        when S_SETUP =>
            if(EF='0' or RDEN='0') then
                wen_a      <= '0';
                sloe_a <= '1';
                slrd_a <= '1';
                state  <= S_IDLE;
            else
                wen_a      <= '0';
                sloe_a <= '0';
                slrd_a <= '1';
                state  <= S_READ;
            end if;
        when S_READ =>
            if(EF='0' or FULL='1' or RDEN = '0') then
                sloe_a <= '0';
                slrd_a <= '1';
                wen_a      <= '0';
                if(FULL='1' and EF='1') then
                    state   <= S_STALL;
                else
                    state <= S_EXIT;
                end if;
            else
                sloe_a <= '0';
                slrd_a <= '0';
                wen_a      <= '1';
            end if;
        when S_EXIT =>
            sloe_a    <= '1';
            slrd_a    <= '1';
            wen_a     <= '0';
            state     <= S_IDLE;
        when S_STALL =>
            if(FULL='0' and RDEN='1' and EF='1') then
                wen_a  <= '1';
                sloe_a <= '0';
                slrd_a <= '0';
                state  <= S_READ;
            elsif(RDEN='0' or EF='0') then
                wen_a      <= '0';
                sloe_a <= '1';
                slrd_a <= '1';
                state  <= S_IDLE;
            else
                sloe_a <= '0';
                slrd_a <= '1';
                wen_a  <= '0';
            end if;
        when others =>
            state     <= S_IDLE;
            sloe_a    <= '1';
            slrd_a    <= '1';
            wen_a     <= '0';
    end case;
  end if;
 end if;
end process;
end CyRead_rtl;
```

1.5 Bus Write

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CyWrite is
    Port ( IFCLK : in  STD_LOGIC;
           WREN : in  STD_LOGIC;
           BUSY : out  STD_LOGIC;
           REN : out  STD_LOGIC;
           EMPTY : in  STD_LOGIC;
           FF : in  STD_LOGIC;
              EF : in STD_LOGIC;
           SLWR : out  STD_LOGIC;
           PKTEND : out  STD_LOGIC;
           RESET : in  STD_LOGIC);
end CyWrite;

architecture CyWrite_rtl of CyWrite is
    type SM_States is (S_IDLE, S_SETUP, S_WRITE, S_EXIT, S_STALL);

    signal state            : SM_States := S_IDLE;
    signal pktend_a     : STD_LOGIC := '1';
    signal slwr_a           : STD_LOGIC := '1';
    signal ren_a            : STD_LOGIC := '0';
    signal sent_bytes       : STD_LOGIC_VECTOR(8 downto 0);
    signal busy_a           : std_logic := '0';
begin
    BUSY <= busy_a;
    SLWR <= slwr_a;
    PKTEND <= pktend_a;
    REN <= ren_a and FF;

    process(IFCLK, RESET) begin
        if(rising_edge(IFCLK)) then
         if(RESET='1') then
            if((ren_a='0') and (slwr_a='1'))then
               pktend_a     <= '1';
               state            <= S_IDLE;
            end if;
            if(ren_a='0') then
               slwr_a       <= '1';
               if(slwr_a='0')then
                  pktend_a      <= '0';
               end if;
            end if;
            ren_a           <= '0';
            sent_bytes  <= (others=>'0');
         else
            case state is
               when S_IDLE =>
                  sent_bytes    <= (others=>'0');
                  ren_a             <= '0';
                  slwr_a            <= '1';
                  pktend_a          <= '1';
                  if(WREN='1' and EMPTY='0' and FF='1') then
                     busy_a         <= '1';
                     state              <= S_SETUP;
                  else
                     busy_a         <= '0';
                  end if;
               when S_SETUP =>
```

```
            sent_bytes    <= (others=>'0');
            slwr_a    <= '1';
            pktend_a  <= '1';
            if(FF='0' or WREN='0') then
               ren_a      <= '0';
               state      <= S_IDLE;
            else
               ren_a      <= '1';
               state      <= S_WRITE;
            end if;
         when S_WRITE =>
            if(slwr_a='0') then
               sent_bytes <= sent_bytes + '1';
               if(FF='1' and EMPTY='0' and WREN='1') then
                  ren_a           <= '1';
                  slwr_a      <= '0';
                  pktend_a        <= '1';
               else
                  ren_a       <= '0';
                  if(FF='0') then
                     slwr_a   <= '1';
                     state    <= S_EXIT;
                  elsif(EMPTY='1') then
                     slwr_a   <= '1';
                     if(sent_bytes="111111111") then
                        state <= S_EXIT;
                     else
                        state <= S_STALL;
                     end if;
                  else
                     state    <= S_EXIT;
                  end if;
               end if;
            else
               slwr_a <= '0';
            end if;
         when S_EXIT =>
            ren_a              <= '0';
            slwr_a             <= '1';
            if(slwr_a='1') then
               if(sent_bytes="000000000") then
                  pktend_a         <= '1';
               else
                  pktend_a         <= '0';
               end if;
            else
               if(sent_bytes="111111111") then
                  pktend_a         <= '1';
               else
                  pktend_a         <= '0';
               end if;
            end if;
            state              <= S_IDLE;
         when S_STALL =>
            slwr_a    <= '1';
            if(EMPTY='0' and WREN='1' and FF='1') then
               ren_a      <= '1';
               state      <= S_WRITE;
            else
               ren_a      <= '0';
               if(WREN='0') then
                  state   <= S_EXIT;
               end if;
            end if;
```

```
            when others =>
               state     <= S_IDLE;
               ren_a     <= '0';
               slwr_a    <= '1';
               pktend_a  <= '1';
         end case;
      end if;
   end if;
end process;
end CyWrite_rtl;
```

3 MMC host detailed schematic

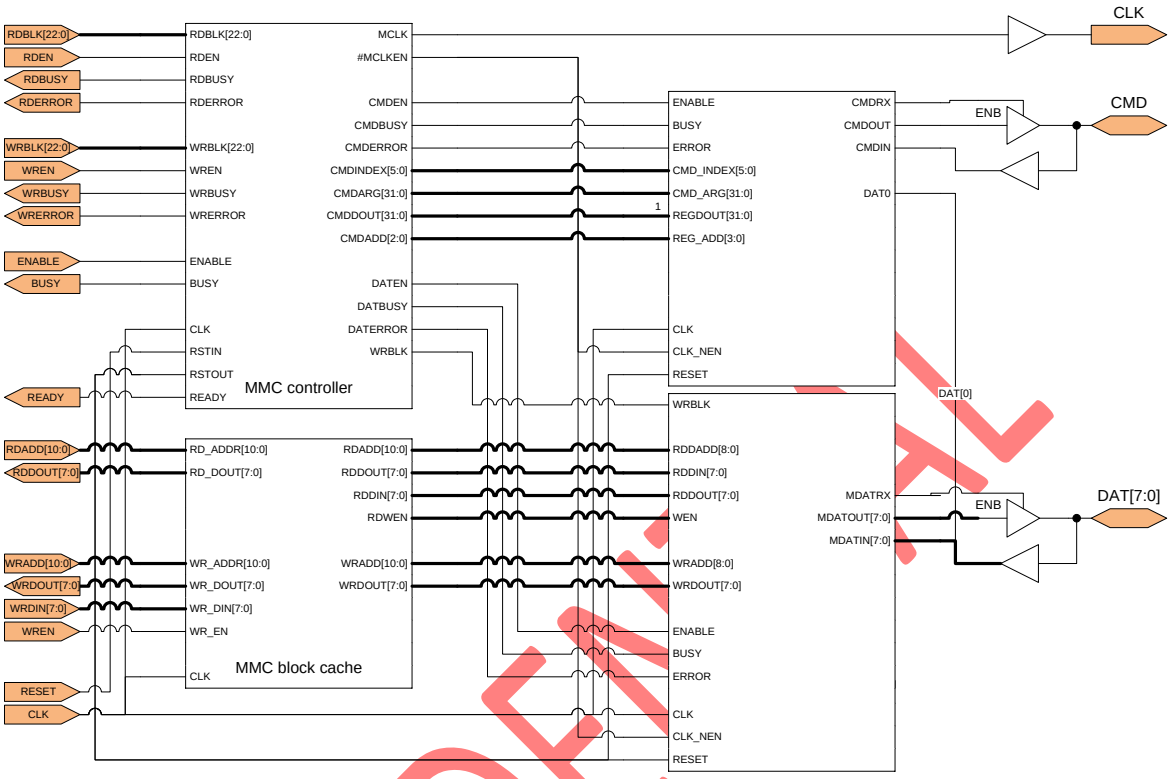


Figure 2 - MMC host schematic

3.1 MMC Host top model

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity MmcDriver is
    Port(
        -- Driver control signals
        RDBLK       : IN  STD_LOGIC_VECTOR(22 downto 0);
        RDEN        : IN  STD_LOGIC;
        RDBUSY  : OUT STD_LOGIC;
        RDERROR : OUT STD_LOGIC;
        WRBLK       : IN  STD_LOGIC_VECTOR(22 downto 0);
        WREN        : IN  STD_LOGIC;
        WRBUSY  : OUT STD_LOGIC;
        WRERROR : OUT STD_LOGIC;
        ENABLE  : in  STD_LOGIC;
        BUSY        : out STD_LOGIC;
        -- Sector memory interface signals
        WRDIN       : IN  STD_LOGIC_VECTOR(7 downto 0);
        WRDOUT  : OUT STD_LOGIC_VECTOR(7 downto 0);
        WRADD       : IN  STD_LOGIC_VECTOR(8 downto 0);
        WEN     : IN  STD_LOGIC;
        RDDOUT  : OUT STD_LOGIC_VECTOR(7 downto 0);
        RDADD       : IN  STD_LOGIC_VECTOR(8 downto 0);
        -- MMC card interface signals
        MCLK        : out STD_LOGIC;
        MCMD        : inout STD_LOGIC;
        MDAT        : inout STD_LOGIC_VECTOR(7 downto 0);
        -- Clock and Reset signals
        READY       : OUT STD_LOGIC;
        GCLK        : IN  STD_LOGIC;
        RESET       : in  STD_LOGIC);
end MmcDriver;

architecture Behavioral of MmcDriver is
    signal sClkIn           : std_logic;
    attribute buffer_type: string;
    attribute buffer_type of sClkIn: signal is "bufgp";--
?{bufgdll|ibufg|bufgp|ibuf|bufr|none}?;

    signal sMclk    : std_logic;
    signal sMclkOut : std_logic;
    signal sNotMclkEn   : std_logic;

    signal sCmdIn   : std_logic;
    signal sCmdOut  : std_logic;
    signal sCmdRx   : std_logic;

    signal sDatIn   : std_logic_vector(7 downto 0);
    signal sDatOut  : std_logic_vector(7 downto 0);
    signal sDatRx   : std_logic;

    COMPONENT mmcSignalSyncro
    GENERIC(
        WIDTH : POSITIVE := 8);
    PORT(
        DIN : IN  std_logic_vector(WIDTH-1 downto 0);
        DOUT    : OUT std_logic_vector(WIDTH-1 downto 0);
```

```
    CLKI    : IN  std_logic;
    CLKO    : IN  std_logic;
    RESET   : IN  std_logic
    );
END COMPONENT;

COMPONENT mmcSingleSignalSyncro
PORT(
    DIN : IN  std_logic;
    DOUT    : OUT std_logic;
    CLKI    : IN  std_logic;
    CLKO    : IN  std_logic;
    RESET   : IN  std_logic
    );
END COMPONENT;

signal sRdBlk       : std_logic_vector(22 downto 0);
signal sWrBlk       : std_logic_vector(22 downto 0);
signal sEnble       : std_logic;
signal sRdEn        : std_logic;
signal sWrEn        : std_logic;
signal sBusy        : std_logic;
signal sRdBusy      : std_logic;
signal sWrBusy      : std_logic;
signal sRdError : std_logic;
signal sWrError : std_logic;

signal sCtlI        : std_logic_vector(2 downto 0);
signal sCtlO        : std_logic_vector(1 downto 0);

COMPONENT DP_2048x8
port(
    clka    : IN  std_logic;
    dina    : IN  std_logic_VECTOR(7 downto 0);
    addra   : IN  std_logic_VECTOR(10 downto 0);
    wea : IN  std_logic_VECTOR(0 downto 0);
    douta   : OUT std_logic_VECTOR(7 downto 0);
    clkb    : IN  std_logic;
    dinb    : IN  std_logic_VECTOR(7 downto 0);
    addrb   : IN  std_logic_VECTOR(10 downto 0);
    web : IN  std_logic_VECTOR(0 downto 0);
    doutb   : OUT std_logic_VECTOR(7 downto 0));
END COMPONENT;

signal sRdMemDin        : std_logic_vector(7 downto 0);
signal sRdMemDout       : std_logic_vector(7 downto 0);
signal sRdMemAdd        : std_logic_vector(8 downto 0);
signal sRdMemWen        : std_logic;

signal sWrMemDout       : std_logic_vector(7 downto 0);
signal sWrMemAdd        : std_logic_vector(8 downto 0);

COMPONENT mmcFsm
    PORT(
    -- DRIVER CONTROL SIGNALS
    RDBLK       : IN  STD_LOGIC_VECTOR(22 DOWNTO 0);
    RDEN        : IN  STD_LOGIC;
    RDBUSY  : OUT STD_LOGIC;
    RDERROR : OUT STD_LOGIC;
    WRBLK       : IN  STD_LOGIC_VECTOR(22 DOWNTO 0);
    WREN        : IN  STD_LOGIC;
    WRBUSY  : OUT STD_LOGIC;
    WRERROR : OUT STD_LOGIC;
    ENABLE  : IN  STD_LOGIC;
```

```
    BUSY         : OUT STD_LOGIC;
    -- MMC CARD INTERFACE SIGNALS
    MCLK         : OUT STD_LOGIC;
    MCLK_NEN     : OUT STD_LOGIC;
    -- CMD FSM INTERFACE SIGNALS
    CMD_ENABLE   : OUT STD_LOGIC;
    CMD_BUSY         : IN  STD_LOGIC;
    CMD_ERROR    : IN  STD_LOGIC;
    CMD_INDEX    : OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
    CMD_ARG      : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    CMD_REGDOUT  : in  STD_LOGIC_VECTOR(31 downto 0);
    CMD_REGADD   : out STD_LOGIC_VECTOR(3 downto 0);
    -- DAT FSM INTERFACE SIGNALS
    DAT_ENABLE   : OUT STD_LOGIC;
    DAT_WRBLK    : OUT STD_LOGIC;
    DAT_BUSY         : IN  STD_LOGIC;
    DAT_ERROR    : IN  STD_LOGIC;
    DAT_BMODE    : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);

    -- CLOCK AND RESET SIGNALS
    READY        : OUT STD_LOGIC;
    CLK50MHz     : IN  STD_LOGIC;
    RESETIN : IN  STD_LOGIC;
    RESETOUT     : OUT STD_LOGIC);
END COMPONENT;

COMPONENT CmdHandler
Port(
    CMDIN            : in  STD_LOGIC;
    CMDOUT       : out STD_LOGIC;
    CMDRX            : out STD_LOGIC;
    DAT0             : in  STD_LOGIC;
    CMD_INDEX    : in  STD_LOGIC_VECTOR(5 downto 0);
    CMD_ARG      : in  STD_LOGIC_VECTOR(31 downto 0);
    REGDOUT      : out STD_LOGIC_VECTOR(31 downto 0);
    REGADD       : in  STD_LOGIC_VECTOR(3 downto 0);
    ENABLE       : in  STD_LOGIC;
    BUSY             : out STD_LOGIC;
    ERROR            : out STD_LOGIC;
    RESET            : in  STD_LOGIC;
    CLK_NEN      : in  STD_LOGIC;
    CLK              : in  STD_LOGIC);
END COMPONENT;

signal sCmdIndex    : std_logic_vector(5 downto 0);
signal sCmdArg      : std_logic_vector(31 downto 0);
signal sCmdRegAdd   : std_logic_vector(3 downto 0);
signal sCmdRegDout: std_logic_vector(31 downto 0);
signal sCmdEnable   : std_logic;
signal sCmdBusy : std_logic;
signal sCmdError    : std_logic;

COMPONENT Mmc1bitInterface
Port(
    ENABLE  : in  STD_LOGIC;
    WRBLK        : in  STD_LOGIC;
    BUSY         : out STD_LOGIC;
    ERROR        : out STD_LOGIC;
    RDDOUT   : out STD_LOGIC_VECTOR(7 downto 0);
    RDDIN        : in  STD_LOGIC_VECTOR(7 downto 0);
    RDADD        : out STD_LOGIC_VECTOR(8 downto 0);
    WEN      : out STD_LOGIC;
    WRDIN        : in  STD_LOGIC_VECTOR(7 downto 0);
    WRADD        : out STD_LOGIC_VECTOR(8 downto 0);
```

```
        MDATIN  : in  STD_LOGIC_VECTOR(7 downto 0);
        MDATOUT : out STD_LOGIC_VECTOR(7 downto 0);
        MDATRX  : out STD_LOGIC;
        CLK     : in  STD_LOGIC;
        CLK_NEN : in  STD_LOGIC;
        RESET       : in  STD_LOGIC);
    END COMPONENT;

    signal sDatEnable   : std_logic;
    signal sDatBusy : std_logic;
    signal sDatWrBlk    : std_logic;
    signal sDatError    : std_logic;
    signal sDatBmode    : std_logic_vector(1 downto 0);
    signal sGlobalReset : std_logic;
    signal sRdWenA          : std_logic_vector(0 downto 0);
    signal sRdWenB          : std_logic_vector(0 downto 0);
    signal sRdAddA          : std_logic_vector(10 downto 0);
    signal sRdAddB          : std_logic_vector(10 downto 0);
    signal sWrWenA          : std_logic_vector(0 downto 0);
    signal sWrWenB          : std_logic_vector(0 downto 0);
    signal sWrAddA          : std_logic_vector(10 downto 0);
    signal sWrAddB          : std_logic_vector(10 downto 0);
begin
    sClkIn  <= GCLK;

    sMclkOut    <= sMclk when (sNotMclkEn='0') else (sMclkOut);
    MCLK        <= sMclkOut;
    MCMD        <= sCmdOut when (sCmdRx='0') else ('Z');
    sCmdIn  <= sCmdOut when (sCmdRx='0') else MCMD;
    MDAT        <= sDatOut when (sDatRx='0') else (others=>'Z');
    sDatIn  <= sDatOut when (sDatRx='0') else MDAT;
    sRdBlk  <= RDBLK;
    sWrBlk  <= WRBLK;
    sEnble  <= ENABLE;
    BUSY    <= sBusy;
    sRdEn   <= RDEN;
    RDBUSY  <= sRdBusy;
    RDERROR <= sRdError;
    sWrEn   <= WREN;
    WRBUSY  <= sWrBusy;
    WRERROR <= sWrError;
    sRdWenA(0)  <= sRdMemWen;
    sRdWenB(0)  <= '0';
    sRdAddA <= "00" & sRdMemAdd;
    sRdAddB <= "00" & RDADD;

    URDRAM: DP_2048x8 PORT MAP (
        clka    => sClkIn,
        dina    => sRdMemDin,
        addra   => sRdAddA,
        wea => sRdWenA,
        douta   => sRdMemDout,
        clkb    => sClkIn,
        dinb    => (others=>'0'),
        addrb   => sRdAddB,
        web => sRdWenB,
        doutb   => RDDOUT
    );

    sWrWenA(0)  <= WEN;
    sWrWenB(0)  <= '0';
    sWrAddA <= "00" & WRADD;
    sWrAddB <= "00" & sWrMemAdd;
```

```
UWRRAM: DP_2048x8 PORT MAP (
    clka    => sClkIn,
    dina    => WRDIN,
    addra   => sWrAddA,
    wea => sWrWenA,
    douta   => WRDOUT,
    clkb    => sClkIn,
    dinb    => (others=>'0'),
    addrb   => sWrAddB,
    web => sWrWenB,
    doutb   => sWrMemDout
);

UMFSM: mmcFsm PORT MAP(
    -- DRIVER CONTROL SIGNALS
    RDBLK => sRdBlk,
    RDEN => sRdEn,
    RDBUSY => sRdBusy,
    RDERROR => sRdError,
    WRBLK => sWrBlk,
    WREN => sWrEn,
    WRBUSY => sWrBusy,
    WRERROR => sWrError,
    ENABLE => sEnble,
    BUSY => sBusy,
    MCLK => sMclk,
    MCLK_NEN => sNotMclkEn,
    CMD_ENABLE => sCmdEnable,
    CMD_BUSY => sCmdBusy,
    CMD_ERROR => sCmdError,
    CMD_INDEX => sCmdIndex,
    CMD_ARG => sCmdArg,
    CMD_REGDOUT => sCmdRegDout,
    CMD_REGADD => sCmdRegAdd,
    DAT_ENABLE => sDatEnable,
    DAT_WRBLK => sDatWrBlk,
    DAT_BUSY => sDatBusy,
    DAT_ERROR => sDatError,
    DAT_BMODE => sDatBmode,
    READY => READY,
    CLK50MHz => sClkIn,
    RESETIN => RESET,
    RESETOUT => sGlobalReset
);

UCMD: CmdHandler PORT MAP(
    CMDIN => sCmdIn,
    CMDOUT => sCmdOut,
    CMDRX => sCmdRx,
    DAT0 => sDatIn(0),
    CMD_INDEX => sCmdIndex,
    CMD_ARG => sCmdArg,
    REGDOUT => sCmdRegDout,
    REGADD => sCmdRegAdd,
    ENABLE => sCmdEnable,
    BUSY => sCmdBusy,
    ERROR => sCmdError,
    CLK_NEN => sNotMclkEn,
    CLK => sMclk,
    RESET => sGlobalReset
);

UDAT: Mmc1bitInterface PORT MAP(
    ENABLE => sDatEnable,
```

```
        WRBLK => sDatWrBlk,
        BUSY => sDatBusy,
        ERROR => sDatError,
        RDDOUT => sRdMemDin,
        RDDIN => sRdMemDout,
        RDADD => sRdMemAdd,
        WEN => sRdMemWen,
        WRDIN => sWrMemDout,
        WRADD => sWrMemAdd,
        MDATIN => sDatIn,
        MDATOUT => sDatOut,
        MDATRX => sDatRx,
        CLK => sMclk,
        CLK_NEN => sNotMclkEn,
        RESET => sGlobalReset
    );
end Behavioral;
```

3.2 CRC Lib

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package PCK_CRC_CCITT is
    -- polynomial: (0 3 7)
    -- data width: 1
    function nextCRC7_D1(Data: std_logic; CRC: std_logic_vector(6 downto 0)) return
std_logic_vector;
    -- polynomial: (0 5 12 16)
    -- data width: 1
    function nextCRC16_D1(Data: std_logic;  CRC: std_logic_vector(15 downto 0)) return
std_logic_vector;
end PCK_CRC_CCITT;

package body PCK_CRC_CCITT is
    function nextCRC7_D1 (Data: std_logic; CRC: std_logic_vector(6 downto 0))
        return std_logic_vector is
        variable D: std_logic_vector(0 downto 0);
        variable C: std_logic_vector(6 downto 0);
        variable NewCRC: std_logic_vector(6 downto 0);
    begin
        D(0) := Data;
        C := CRC;
        NewCRC(0) := D(0) xor C(6);
        NewCRC(1) := C(0);
        NewCRC(2) := C(1);
        NewCRC(3) := D(0) xor C(2) xor C(6);
        NewCRC(4) := C(3);
        NewCRC(5) := C(4);
        NewCRC(6) := C(5);
        return NewCRC;
    end nextCRC7_D1;

    function nextCRC16_D1 (Data: std_logic; CRC: std_logic_vector(15 downto 0)) return
std_logic_vector is
        variable D: std_logic_vector(0 downto 0);
        variable C: std_logic_vector(15 downto 0);
        variable NewCRC: std_logic_vector(15 downto 0);
    begin
        D(0) := Data;
        C := CRC;
        NewCRC(0) := D(0) xor C(15);
        NewCRC(1) := C(0);
        NewCRC(2) := C(1);
        NewCRC(3) := C(2);
        NewCRC(4) := C(3);
        NewCRC(5) := D(0) xor C(4) xor C(15);
        NewCRC(6) := C(5);
        NewCRC(7) := C(6);
        NewCRC(8) := C(7);
        NewCRC(9) := C(8);
        NewCRC(10) := C(9);
        NewCRC(11) := C(10);
        NewCRC(12) := D(0) xor C(11) xor C(15);
        NewCRC(13) := C(12);
        NewCRC(14) := C(13);
        NewCRC(15) := C(14);
        return NewCRC;
    end nextCRC16_D1;
end PCK_CRC_CCITT;
```

3.3 CRC 7

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.PCK_CRC_CCITT.all;

entity CRC7 is
    Generic(
        POS_EDGE_TRIG   : boolean := true;
        INITIAL_VALUE   : std_logic_vector(6 downto 0) := (others=>'0'));
    Port(
        CLK     : in  STD_LOGIC;
        DIN     : in  STD_LOGIC;
        DOUT        : out STD_LOGIC;
        CRC7        : out STD_LOGIC_VECTOR (6 downto 0);
        CLEAR       : in  STD_LOGIC;
        ENABLE  : in  STD_LOGIC);
end CRC7;

architecture Behavioral of CRC7 is
    signal sClkIn           : std_logic;
    attribute buffer_type: string;
    attribute buffer_type of sClkIn: signal is "bufgp";
    signal regCRC   : std_logic_vector(6 downto 0) := INITIAL_VALUE;
begin

    sClkIn  <= CLK;
    DOUT    <= regCRC(6);
    CRC7    <= regCRC;
    POS_EDGE_TIRGGER:
    if (POS_EDGE_TRIG=true) generate begin
        process(sClkIn, CLEAR) begin
            if(CLEAR='1') then
                regCRC  <= INITIAL_VALUE;
            else
                if(rising_edge(sClkIn)) then
                    if(ENABLE='1') then
                        regCRC  <= nextCRC7_D1(DIN, regCRC);
                    end if;
                end if;
            end if;
        end process;
    end generate;

    NEG_EDGE_TIRGGER:
    if (POS_EDGE_TRIG=false) generate begin
        process(sClkIn, CLEAR) begin
            if(CLEAR='1') then
                regCRC  <= INITIAL_VALUE;
            else
                if(falling_edge(sClkIn)) then
                    if(ENABLE='1') then
                        regCRC  <= nextCRC7_D1(DIN, regCRC);
                    end if;
                end if;
            end if;
        end process;
    end generate;

end Behavioral;
```

3.4 CRC16

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.PCK_CRC_CCITT.all;

entity CRC16 is
    Generic(
        INITIAL_VALUE   : std_logic_vector(15 downto 0) := (others=>'0'));
    Port(
        CLK     : in  STD_LOGIC;
        DIN     : in  STD_LOGIC;
        DOUT        : out STD_LOGIC;
        CRC16       : out STD_LOGIC_VECTOR (15 downto 0);
        CLEAR       : in  STD_LOGIC;
        ENABLE  : in  STD_LOGIC);
end CRC16;

architecture Behavioral of CRC16 is
    signal sClkIn           : std_logic;
    attribute buffer_type: string;
    attribute buffer_type of sClkIn: signal is "bufgp";
    signal regCRC   : std_logic_vector(15 downto 0) := INITIAL_VALUE;
begin
    sClkIn  <= CLK;

    DOUT    <= regCRC(15);
    CRC16   <= regCRC;
    process(sClkIn, CLEAR) begin
        if(CLEAR='1') then
            regCRC  <= INITIAL_VALUE;
        else
            if(rising_edge(sClkIn)) then
                if(ENABLE='1') then
                    regCRC  <= nextCRC16_D1(DIN, regCRC);
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

3.5 Shift register

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ShiftRegisterIn is
    Generic(
        DATAWIDTH   : positive := 8);
    Port(
        SDIN    : in  STD_LOGIC;
        PDOUT   : out STD_LOGIC_VECTOR(DATAWIDTH-1 downto 0);
        ENABLE: in  STD_LOGIC;
        RESET   : in  STD_LOGIC;
        CLK : in  STD_LOGIC);
end ShiftRegisterIn;

architecture Behavioral of ShiftRegisterIn is
    signal sClkIn           : std_logic;
    attribute buffer_type: string;
    attribute buffer_type of sClkIn: signal is "bufgp";
    signal regPdout : std_logic_vector(DATAWIDTH-1 downto 0) := (others=>'0');
begin
    sClkIn  <= CLK;
    PDOUT   <= regPdout;
    process(sClkIn, RESET) begin
        if(RESET='1') then
            regPdout    <= (others=>'0');
        else
            if(rising_edge(sClkIn)) then
                if(ENABLE='1') then
                    regPdout    <= regPdout(DATAWIDTH-2 downto 0) & SDIN;
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

3.6 Command handler

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CmdHandler is
    Port(
        CMDIN           : in  STD_LOGIC;
        CMDOUT      : out STD_LOGIC;
        CMDRX           : out STD_LOGIC;
        DAT0            : in  STD_LOGIC;
        CMD_INDEX   : in  STD_LOGIC_VECTOR(5 downto 0);
        CMD_ARG     : in  STD_LOGIC_VECTOR(31 downto 0);
        REGDOUT     : out STD_LOGIC_VECTOR(31 downto 0);
        REGADD      : in  STD_LOGIC_VECTOR(3 downto 0);
        ENABLE      : in  STD_LOGIC;
        BUSY            : out STD_LOGIC;
        ERROR           : out STD_LOGIC;
        RESET           : in  STD_LOGIC;
        CLK_NEN     : in  STD_LOGIC;
        CLK         : in  STD_LOGIC);
end CmdHandler;

architecture Behavioral of CmdHandler is
    signal sClkIn           : std_logic;
    attribute buffer_type: string;
    attribute buffer_type of sClkIn: signal is "bufgp";
    component DP_DRam_512x8
    Generic(
        WIDTH : positive := 8;
        DEPTH : positive := 9);
    Port(
        a       : in  STD_LOGIC_VECTOR(DEPTH-1 downto 0);
        d       : in  STD_LOGIC_VECTOR(WIDTH-1 downto 0);
        spo : out STD_LOGIC_VECTOR(WIDTH-1 downto 0);
        we      : in  STD_LOGIC;

        dpra    : in  STD_LOGIC_VECTOR(DEPTH-1 downto 0);
        dpo : out STD_LOGIC_VECTOR(WIDTH-1 downto 0);
        clk     : in  STD_LOGIC);
    end component;

    signal sRdMemDin        : std_logic_vector(31 downto 0);
    signal sRdMemAdd        : std_logic_vector(3 downto 0);
    signal sRdMemWen        : std_logic;
    signal sCmdIn           : std_logic;
    signal sCmdInB          : std_logic;
    signal sCmdOut          : std_logic := '1';
    signal sCmdRx           : std_logic := '1';
    signal sCmdIndex    : std_logic_vector(5 downto 0)  := (others=>'0');
    signal sCmdArg      : std_logic_vector(31 downto 0) := (others=>'0');
    type ResponseType is (R0, R1, R1b, R2, R3, R4, R5);
    signal sResponse    : ResponseType;
    signal sRespCnt : std_logic_vector(7 downto 0)  := (others=>'0');
    signal sRespLen : std_logic_vector(7 downto 0)  := (others=>'0');

    COMPONENT CRC7
    Generic(
        POS_EDGE_TRIG   : boolean := true;
        INITIAL_VALUE   : std_logic_vector(6 downto 0) := (others=>'0'));
    PORT(
        CLK : IN std_logic;
```

```
        DIN : IN std_logic;
        CLEAR : IN std_logic;
        ENABLE : IN std_logic;
        DOUT : OUT std_logic;
        CRC7 : OUT std_logic_vector(6 downto 0)
        );
    END COMPONENT;

    signal sCrcDin  : std_logic;

    COMPONENT ShiftRegisterIn
    GENERIC(
        DATAWIDTH   : positive);
    PORT(
        SDIN : IN std_logic;
        ENABLE : IN std_logic;
        RESET : IN std_logic;
        CLK : IN std_logic;
        PDOUT : OUT std_logic_vector(DATAWIDTH-1 downto 0)
        );
    END COMPONENT;

    signal sSrEnable    : std_logic;
    signal sSrEnGlob    : std_logic;
    signal sSrPdout : std_logic_vector(135 downto 0);
    signal sCrc         : std_logic_vector(6 downto 0);
    signal sCrcDout : std_logic;
    signal sCrcEnable   : std_logic;
    signal sCrcEnGlob   : std_logic;
    signal sCrcClear    : std_logic;
    type SM_States is (S_IDLE, S_WAIT_NRC_CYCLES,
                       S_SND_START_BIT, S_SND_TRANS_BIT, S_SND_CMD_INDEX,
S_SND_CMD_ARG, S_SND_CRC, S_SND_END_BIT,
                       S_WAIT_BUSY_DAT0, S_POLL_BUSY_DAT0,
                       S_GET_START_BIT, S_GET_TRANS_BIT, S_GET_CONTENT,
S_GET_END_BIT,
                       S_PROCESS_RESPONSE,
                       S_EXIT);

    signal state    : SM_States := S_IDLE;
    signal sBitCnt  : std_logic_vector(5 downto 0);
begin
    sClkIn  <= CLK;
    URDRAM: DP_DRam_512x8
    GENERIC MAP(
        WIDTH => 32, --32 bit words
        DEPTH => 4) --16 words
    PORT MAP(
        a => sRdMemAdd,
        d => sRdMemDin,
        dpra => REGADD,
        clk => sClkIn,
        we => sRdMemWen,
        spo => open,--sRdMemDout,
        dpo => REGDOUT
    );

    sResponse   <= R0 when (sCmdIndex=conv_std_logic_vector(0,5)) else
                   R3 when (sCmdIndex=conv_std_logic_vector(1,5)) else
                   R2 when (sCmdIndex=conv_std_logic_vector(2,5)) else
                   R1 when (sCmdIndex=conv_std_logic_vector(3,5)) else
                   R0 when (sCmdIndex=conv_std_logic_vector(4,5)) else
                   R1b when (sCmdIndex=conv_std_logic_vector(6,5)) else
                   R1b when (sCmdIndex=conv_std_logic_vector(7,5)) else
```

```
                    R1 when (sCmdIndex=conv_std_logic_vector(8,5)) else
                    R2 when (sCmdIndex=conv_std_logic_vector(9,5)) else
                    R2 when (sCmdIndex=conv_std_logic_vector(10,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(11,5)) else
                    R1b when (sCmdIndex=conv_std_logic_vector(12,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(13,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(14,5)) else
                    R0 when (sCmdIndex=conv_std_logic_vector(15,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(19,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(16,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(17,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(18,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(20,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(23,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(24,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(25,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(26,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(27,5)) else
                    R1b when (sCmdIndex=conv_std_logic_vector(28,5)) else
                    R1b when (sCmdIndex=conv_std_logic_vector(29,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(30,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(35,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(36,5)) else
                    R1b when (sCmdIndex=conv_std_logic_vector(38,5)) else
                    R4 when (sCmdIndex=conv_std_logic_vector(39,5)) else
                    R5 when (sCmdIndex=conv_std_logic_vector(40,5)) else
                    R1b when (sCmdIndex=conv_std_logic_vector(42,5)) else
                    R1 when (sCmdIndex=conv_std_logic_vector(55,5)) else
                    R1b when (sCmdIndex=conv_std_logic_vector(56,5)) else
                    R0;

sRespLen    <= conv_std_logic_vector(0, 8) when (sResponse=R0) else
               conv_std_logic_vector(134, 8) when (sResponse=R2) else
               conv_std_logic_vector(46, 8);

sCmdInB <= CMDIN;
CMDOUT  <= sCmdOut;
CMDRX       <= sCmdRx;

process(sClkIn) begin
    if(rising_edge(sClkIn) and (CLK_NEN='0')) then
        sCmdIn  <= sCmdInB;
    end if;
end process;

UCRC7: CRC7
GENERIC MAP(
    POS_EDGE_TRIG => true,
    INITIAL_VALUE => (others=>'0'))
PORT MAP(
    CLK => sClkIn,
    DIN => sCrcDin,
    DOUT => sCrcDout,
    CRC7 => sCrc,
    CLEAR => sCrcClear,
    ENABLE => sCrcEnGlob
);

sCrcEnGlob  <= (sCrcEnable and (not CLK_NEN));
sCrcDin <= sCmdInB;

USRIN: ShiftRegisterIn
GENERIC MAP(
    DATAWIDTH => 136)
```

```
PORT MAP(
    SDIN => sCmdIn,
    PDOUT => sSrPdout,
    ENABLE => sSrEnGlob,
    RESET => RESET,
    CLK => sClkIn
);
sSrEnGlob   <= (sSrEnable and (not CLK_NEN));
process(sClkIn, RESET) begin
    if(RESET='1') then
        sCmdOut     <= '1';
        sCmdRx      <= '1';
        sCrcClear   <= '1';
        sCrcEnable  <= '0';
        sSrEnable   <= '0';
        sRdMemWen   <= '0';
        sBitCnt     <= (others=>'0');
        state           <= S_IDLE;
    else
        if((falling_edge(sClkIn)) and (CLK_NEN='0')) then
            case (state) is
                when S_IDLE =>
                    if(ENABLE='1') then
                        sBitCnt     <= conv_std_logic_vector(8,6);
                        sCmdIndex   <= CMD_INDEX;
                        sCmdArg     <= CMD_ARG;
                        sCmdRx      <= '1';
                        sCrcClear   <= '1';
                        sCrcEnable  <= '0';
                        state           <= S_WAIT_NRC_CYCLES;
                    else
                        sCmdOut     <= '1';
                        sCmdRx      <= '1';
                        sCrcClear   <= '1';
                        sCrcEnable  <= '0';
                        sSrEnable   <= '0';
                    end if;
                when S_WAIT_NRC_CYCLES =>
                    if(sBitCnt="000000") then
                        sCmdOut     <= '0';
                        sCmdRx      <= '0';
                        sCrcClear   <= '0';
                        sCrcEnable  <= '1';
                        state           <= S_SND_START_BIT;
                    else
                        sBitCnt <= sBitCnt - '1';
                    end if;
                when S_SND_START_BIT =>
                    sCmdOut <= '1';
                    state       <= S_SND_TRANS_BIT;
                when S_SND_TRANS_BIT =>
                    sBitCnt <= conv_std_logic_vector(4, 6);
                    sCmdOut <= sCmdIndex(5);
                    state       <= S_SND_CMD_INDEX;
                when S_SND_CMD_INDEX =>
                    sCmdOut <= sCmdIndex(conv_integer(sBitCnt));
                    sBitCnt <= sBitCnt - '1';
                    if(sBitCnt="00000") then
                        sBitCnt <= conv_std_logic_vector(31, 6);
                        state       <= S_SND_CMD_ARG;
                    end if;
                when S_SND_CMD_ARG =>
                    sCmdOut <= sCmdArg(conv_integer(sBitCnt));
                    sBitCnt <= sBitCnt - '1';
```

```
        if(sBitCnt="00000") then
            sBitCnt     <= conv_std_logic_vector(6, 6);
            state           <= S_SND_CRC;
        end if;
    when S_SND_CRC =>
        sCmdOut <= sCrcDout;
        sBitCnt <= sBitCnt - '1';
        if(sBitCnt="00000") then
            sCrcEnable  <= '0';
            state   <= S_SND_END_BIT;
        end if;
    when S_SND_END_BIT =>
        sCmdOut <= '1';
        if(sResponse=R0) then
            sCrcClear   <= '1';
            sCrcEnable  <= '0';
            state           <= S_EXIT;
        elsif(sResponse=R1b) then
            sCrcClear   <= '1';
            sCrcEnable  <= '0';
            sBitCnt     <= conv_std_logic_vector(8,6);
            state           <= S_WAIT_BUSY_DAT0;
            sRespCnt        <= (others=>'0');
        else
            sCrcClear   <= '1';
            sCrcEnable  <= '0';
            sBitCnt     <= conv_std_logic_vector(63,6);
            state           <= S_GET_START_BIT;
            sRespCnt        <= (others=>'0');
        end if;
    when S_WAIT_BUSY_DAT0 =>
        sCmdRx  <= '1';
        sBitCnt <= sBitCnt - '1';
        if(sCmdIn='0')then
            sCrcClear   <= '0';
            sCrcEnable  <= '0';
            sSrEnable   <= '1';
            sRespCnt        <= sRespCnt + '1';
            state           <= S_GET_TRANS_BIT;
        elsif(DAT0='0')then
            state   <= S_POLL_BUSY_DAT0;
        elsif(sBitCnt="000000") then
            sBitCnt <= conv_std_logic_vector(63,6);
            state       <= S_GET_START_BIT;
        end if;
    when S_POLL_BUSY_DAT0 =>
        if(sCmdIn='0')then
            sCrcClear   <= '0';
            sCrcEnable  <= '0';
            sSrEnable   <= '1';
            sRespCnt        <= sRespCnt + '1';
            state           <= S_GET_TRANS_BIT;
        elsif(DAT0='1') then
            sBitCnt <= conv_std_logic_vector(63,6);
            state       <= S_GET_START_BIT;
        end if;
    when S_GET_START_BIT =>
        sCmdRx  <= '1';
        if(sCmdIn='0') then
            sCrcClear   <= '0';
            sCrcEnable  <= '0';
            sSrEnable   <= '1';
            sRespCnt        <= sRespCnt + '1';
            state           <= S_GET_TRANS_BIT;
```

```
                else
                    sCrcClear   <= '1';
                    sCrcEnable  <= '0';
                    if(sBitCnt="00000") then
                        ERROR   <= '1';
                        state   <= S_EXIT;
                    else
                        sBitCnt <= sBitCnt - '1';
                    end if;
                end if;
            when S_GET_TRANS_BIT =>
                if(sCmdIn='0') then
                    sRespCnt    <= sRespCnt + '1';
                    state       <= S_GET_CONTENT;
                else
                    ERROR   <= '1';
                    state   <= S_EXIT;
                end if;
            when S_GET_CONTENT =>
                sRespCnt        <= sRespCnt + '1';
                if((sResponse=R1) or (sResponse=R4) or (sResponse=R5))then
                    sCrcEnable  <= '1';
                elsif(sResponse=R2)then
                    if(sRespCnt=conv_std_logic_vector(7,8)) then
                        sCrcEnable  <= '1';
                    end if;
                end if;
                if(sRespCnt=sRespLen) then
                    state           <= S_GET_END_BIT;
                    sCrcEnable  <= '0';
                end if;
            when S_GET_END_BIT =>
                if(((sCrc="0000000") or (sResponse=R3)) and (sCmdIn='1')) then
                    if((sResponse=R1) or (sResponse=R1b))then
                        sRdMemAdd   <= conv_std_logic_vector(0,4);
                        sBitCnt     <= conv_std_logic_vector(1,6);
                    elsif(sResponse=R2) then
                        if((sCmdIndex=conv_std_logic_vector(2,6)) or
(sCmdIndex=conv_std_logic_vector(10,6))) then
                            sRdMemAdd   <= conv_std_logic_vector(2,4);
                            sBitCnt     <= conv_std_logic_vector(4,6);
                        elsif(sCmdIndex=conv_std_logic_vector(9,6)) then
                            sRdMemAdd   <= conv_std_logic_vector(6,4);
                            sBitCnt     <= conv_std_logic_vector(4,6);
                        end if;
                    elsif(sResponse=R3) then
                        sRdMemAdd   <= conv_std_logic_vector(1,4);
                        sBitCnt     <= conv_std_logic_vector(1,6);
                    elsif(sResponse=R4) then
                        sRdMemAdd   <= conv_std_logic_vector(10,4);
                        sBitCnt     <= conv_std_logic_vector(1,6);
                    elsif(sResponse=R5) then
                        sRdMemAdd   <= conv_std_logic_vector(11,4);
                        sBitCnt     <= conv_std_logic_vector(1,6);
                    end if;
                    state   <= S_PROCESS_RESPONSE;
                else
                    ERROR   <= '1';
                    state   <= S_EXIT;
                end if;
            when S_PROCESS_RESPONSE =>
                sSrEnable   <= '0';
                if(sBitCnt="000000")then
                    sRdMemWen   <= '0';
```

```
                            state           <= S_EXIT;
                        else
                            sRdMemWen   <= '1';
                            sRdMemDin   <= sSrPdout((31+(conv_integer(sBitCnt)*8)) downto
(conv_integer(sBitCnt)*8));
                            sBitCnt     <= sBitCnt - '1';
                            if(sRdMemWen='1')then
                                sRdMemAdd   <= sRdMemAdd + '1';
                            end if;
                        end if;
                    when S_EXIT =>
                        sCmdRx      <= '1';
                        sSrEnable   <= '0';
                        if(ENABLE='0') then
                            ERROR   <= '0';
                            state   <= S_IDLE;
                        end if;
                    when others =>
                        sCmdOut     <= '1';
                        sCmdRx      <= '1';
                        sCrcClear   <= '1';
                        sCrcEnable  <= '0';
                        sSrEnable   <= '0';
                        sRdMemWen   <= '0';
                        sBitCnt     <= (others=>'0');
                        state           <= S_IDLE;
                end case;
            end if;
        end if;
    end process;

    BUSY    <= '0' when ((state=S_IDLE) or (state=S_EXIT)) else '1';
end Behavioral;
```

3.7 1bit data interface

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Mmc1bitInterface is
    Port(
        ENABLE  : in  STD_LOGIC;
        WRBLK       : in  STD_LOGIC;
        BUSY        : out STD_LOGIC;
        ERROR       : out STD_LOGIC;
        RDDOUT  : out STD_LOGIC_VECTOR(7 downto 0);
        RDDIN       : in  STD_LOGIC_VECTOR(7 downto 0);
        RDADD       : out STD_LOGIC_VECTOR(8 downto 0);
        WEN     : out STD_LOGIC;
        WRDIN       : in  STD_LOGIC_VECTOR(7 downto 0);
        WRADD       : out STD_LOGIC_VECTOR(8 downto 0);
        MDATIN  : in  STD_LOGIC_VECTOR(7 downto 0);
        MDATOUT : out STD_LOGIC_VECTOR(7 downto 0);
        MDATRX  : out STD_LOGIC;
        CLK     : in  STD_LOGIC;
        CLK_NEN : in  STD_LOGIC;
        RESET       : in  STD_LOGIC);
end Mmc1bitInterface;

architecture Behavioral of Mmc1bitInterface is
    signal sClkIn           : std_logic;
    attribute buffer_type: string;
    attribute buffer_type of sClkIn: signal is "bufgp";

    signal sMdinB   : std_logic;
    signal sMdin    : std_logic;
    signal sMdout   : std_logic;
    signal sMdatRx  : std_logic := '1';

    signal regDin   : std_logic_vector(7 downto 0);
    signal regDout  : std_logic_vector(7 downto 0);
    signal regAdd   : std_logic_vector(8 downto 0);
    signal sWen     : std_logic;
    signal sRen     : std_logic;

    COMPONENT CRC16
    PORT(
        CLK : IN std_logic;
        DIN : IN std_logic;
        CLEAR : IN std_logic;
        ENABLE : IN std_logic;
        DOUT : OUT std_logic;
        CRC16 : OUT std_logic_vector(15 downto 0)
        );
    END COMPONENT;

    COMPONENT ShiftRegisterIn
    GENERIC(
        DATAWIDTH   : positive);
    PORT(
        SDIN : IN std_logic;
        ENABLE : IN std_logic;
        RESET : IN std_logic;
        CLK : IN std_logic;
        PDOUT : OUT std_logic_vector(DATAWIDTH-1 downto 0)
        );
```

```
    END COMPONENT;

    signal sSrEnable    : std_logic;
    signal sSrEnGlob    : std_logic;
    signal sSrPdout : std_logic_vector(7 downto 0);

    signal regCrc       : std_logic_vector(15 downto 0);
    signal sCrcDout : std_logic;
    signal sCrcClear    : std_logic;
    signal sCrcEnable   : std_logic;
    signal sCrcEnGlob   : std_logic;

    type SM_States is (S_IDLE, S_WAIT_NWR_CYCLES,
                            S_SND_START_BIT, S_SND_DATA, S_SND_CRC, S_SND_END_BIT,
                            S_GET_CRC_TOKEN_START_BIT, S_GET_CRC_TOKEN,
S_GET_CRC_TOKEN_END_BIT, S_WAIT_BUSY,
                            S_GET_START_BIT, S_GET_DATA, S_GET_CRC, S_GET_END_BIT,
                            S_EXIT);

    signal state    : SM_States := S_IDLE;

    signal sBitCnt      : std_logic_vector(5 downto 0);
    signal regCrcTok    : std_logic_vector(2 downto 0);

begin
    sClkIn  <= CLK;

    RDADD       <= regAdd;
    RDDOUT  <= regDout;
    WEN     <= sWen;

    WRADD       <= regAdd;
    regDin  <= WRDIN;

    sMdinB      <= MDATIN(0);
    MDATRX      <= sMdatRx;
    MDATOUT(0)  <= sMdout;
    MDATOUT(7 downto 1) <= (others=>'1');

    UCRC: CRC16 PORT MAP(
        CLK => sClkIn,
        DIN => sMdinB,
        DOUT => sCrcDout,
        CRC16 => regCrc,
        CLEAR => sCrcClear,
        ENABLE => sCrcEnGlob
    );
    sCrcEnGlob  <= (sCrcEnable and (not CLK_NEN));

    USRIN: ShiftRegisterIn
    GENERIC MAP(
        DATAWIDTH => 8)
    PORT MAP(
        SDIN => sMdinB,
        PDOUT => sSrPdout,
        ENABLE => sSrEnGlob,
        RESET => RESET,
        CLK => sClkIn
    );
    regDout <= sSrPdout;
    sSrEnGlob   <= (sSrEnable and (not CLK_NEN));

    process(sClkIn, RESET) begin
        if(RESET='1') then
```

```
        sMdin   <= '0';
    else
        if(rising_edge(sClkIn) and (CLK_NEN='0'))then
            sMdin   <= sMdinB;
        end if;
    end if;
end process;

process(sClkIn, RESET) begin
    if(RESET='1') then
        sMdout      <= '1';
        sMdatRx     <= '1';
        sCrcClear   <= '1';
        sCrcEnable  <= '0';
        regAdd      <= (others=>'0');
        sWen            <= '0';
        sSrEnable   <= '0';
        state           <= S_IDLE;
    else
        if((falling_edge(sClkIn)) and (CLK_NEN='0'))then
            case (state) is
                when S_IDLE =>
                    sMdatRx     <= '1';
                    sMdout      <= '1';
                    sCrcClear   <= '1';
                    sCrcEnable  <= '0';
                    regAdd      <= (others=>'0');
                    sSrEnable   <= '0';
                    if((ENABLE='1') and (sMdin='1'))then
                        if(WRBLK='0') then
                            state       <= S_GET_START_BIT;
                        elsif(WRBLK='1') then
                            sBitCnt <= conv_std_logic_vector(2,6);
                            state       <= S_WAIT_NWR_CYCLES;
                        else
                            ERROR       <= '1';
                            state       <= S_EXIT;
                        end if;
                    end if;
                when S_WAIT_NWR_CYCLES =>
                    if(sBitCnt="000000") then
                        sMdout      <= '0';
                        sMdatRx     <= '0';
                        sCrcClear   <= '0';
                        sCrcEnable  <= '1';
                        state           <= S_SND_START_BIT;
                        sBitCnt     <= conv_std_logic_vector(7,6);
                    elsif(sBitCnt/="000000") then
                        sBitCnt <= sBitCnt - '1';
                    end if;
                when S_SND_START_BIT =>
                    sMdout  <= regDin(conv_integer(sBitCnt));
                    sBitCnt <= sBitCnt - '1';
                    state       <= S_SND_DATA;
                when S_SND_DATA =>
                    sMdout  <= regDin(conv_integer(sBitCnt));
                    if((regAdd="111111111") and (sBitCnt="000000")) then
                        sRen        <= '0';
                        sBitCnt <= conv_std_logic_vector(15,6);
                        state       <= S_SND_CRC;
                    else
                        if(sBitCnt="000000") then
                            sRen        <= '1';
                            sBitCnt <= conv_std_logic_vector(7,6);
```

```
                regAdd  <= regAdd + '1';
            else
                sRen        <= '0';
                sBitCnt <= sBitCnt - '1';
            end if;
        end if;
    when S_SND_CRC =>
        sMdout  <= sCrcDout;
        sBitCnt <= sBitCnt - '1';
        if(sBitCnt="000000") then
            sCrcEnable  <= '0';
            state           <= S_SND_END_BIT;
        end if;
    when S_SND_END_BIT =>
        sMdout  <= '1';
        sBitCnt <= conv_std_logic_vector(63,6);
        state       <= S_GET_CRC_TOKEN_START_BIT;
    when S_GET_CRC_TOKEN_START_BIT =>
        sMdatRx <= '1';
        if(sMdin='0') then
            sBitCnt <= conv_std_logic_vector(2,6);
            state   <= S_GET_CRC_TOKEN;
        elsif(sBitCnt="000000")then
            ERROR   <= '1';
            state   <= S_EXIT;
        else
            sBitCnt <= sBitCnt - '1';
        end if;
    when S_GET_CRC_TOKEN =>
        regCrcTok(conv_integer(sBitCnt))    <= sMdin;
        if(sBitCnt="000000")then
            state   <= S_GET_CRC_TOKEN_END_BIT;
        else
            sBitCnt <= sBitCnt - '1';
        end if;
    when S_GET_CRC_TOKEN_END_BIT =>
        if((sMdin='1') and (regCrcTok="010")) then
            state   <= S_WAIT_BUSY;
        else
            ERROR   <= '1';
            state   <= S_EXIT;
        end if;
    when S_WAIT_BUSY =>
        if(sMdin='1') then
            state   <= S_EXIT;
        end if;
    when S_GET_START_BIT =>
        sMdatRx <= '1';
        if(sMdin='0') then
            sCrcClear   <= '0';
            sCrcEnable  <= '1';
            sSrEnable   <= '1';
            sBitCnt     <= conv_std_logic_vector(7,6);
            state           <= S_GET_DATA;
        else
            sCrcClear   <= '1';
            sCrcEnable  <= '0';
            if(ENABLE='0')then
                state   <= S_EXIT;
            end if;
        end if;
    when S_GET_DATA =>
        if(sBitCnt="000000")then
            sBitCnt <= conv_std_logic_vector(7,6);
```

```
                    sWen        <= '1';
                    if(regAdd="111111111")then
                        sBitCnt <= conv_std_logic_vector(15,6);
                        state   <= S_GET_CRC;
                    end if;
                else
                    sBitCnt <= sBitCnt - '1';
                    if(sWen='1') then
                        regAdd  <= regAdd + '1';
                    end if;
                    sWen    <= '0';
                end if;
            when S_GET_CRC =>
                sWen    <= '0';
                if(sBitCnt="000000")then
                    sCrcEnable  <= '0';
                    state           <= S_GET_END_BIT;
                else
                    sBitCnt <= sBitCnt - '1';
                end if;
            when S_GET_END_BIT =>
                if((regCrc=x"0000") and (sMdin='1'))then
                    state   <= S_EXIT;
                else
                    ERROR   <= '1';
                    state   <= S_EXIT;
                end if;
            when S_EXIT =>
                sMdatRx     <= '1';
                if(ENABLE='0') then
                    ERROR   <= '0';
                    state   <= S_IDLE;
                end if;
            when others =>
                sMdout      <= '1';
                sMdatRx     <= '1';
                sCrcClear   <= '1';
                sCrcEnable  <= '0';
                regAdd      <= (others=>'0');
                sWen        <= '0';
                sSrEnable   <= '0';
                state           <= S_IDLE;
            end case;
        end if;
    end if;
end process;

BUSY    <= '0' when ((state=S_IDLE) or (state=S_EXIT))else '1';
end Behavioral;
```

4 Wireless communications detailed schematic

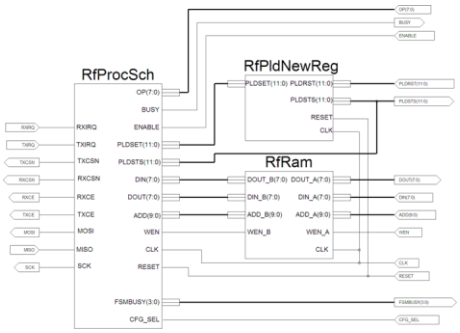


Figure 3 - RF Host

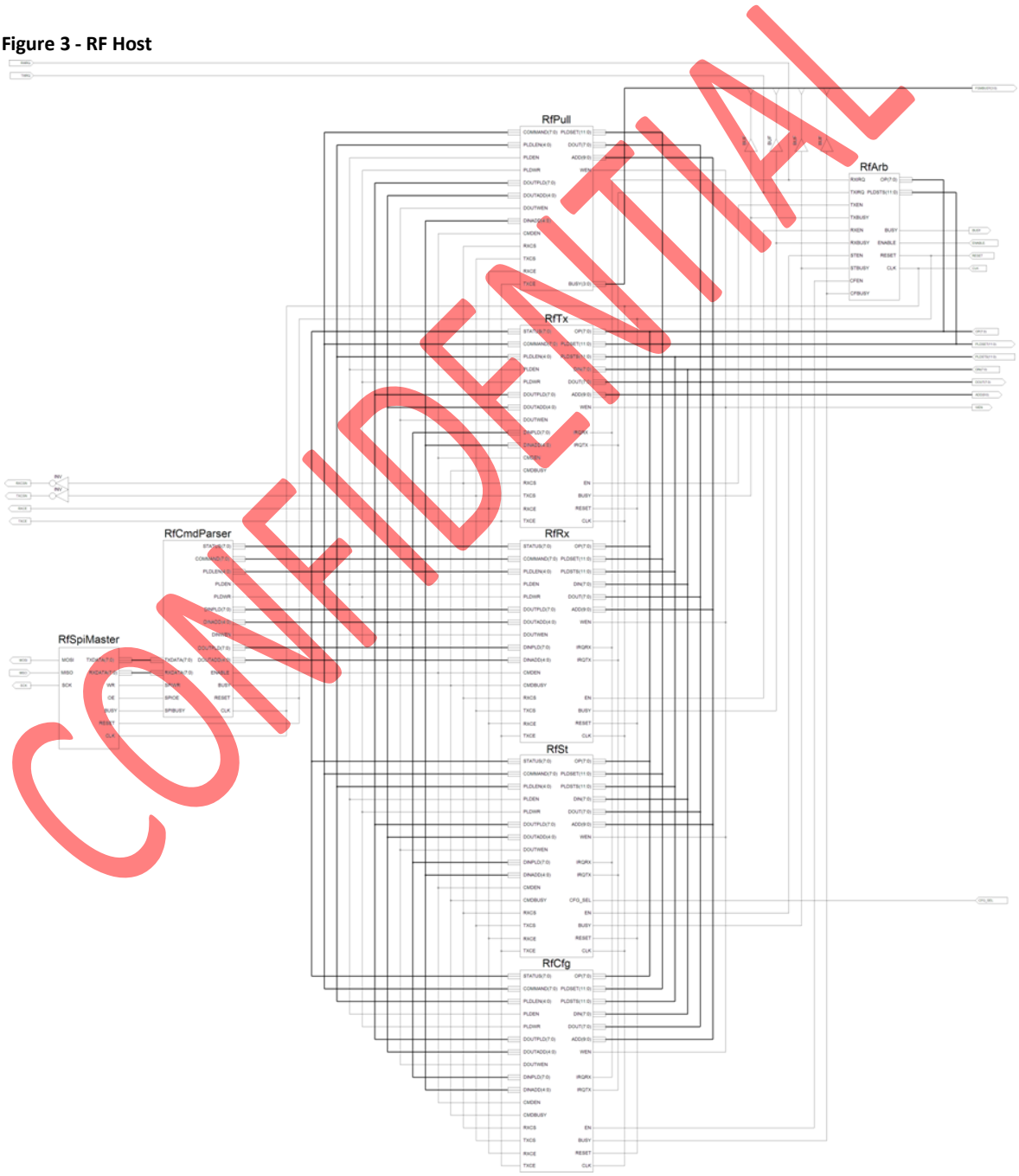


Figure 4 - RF processor

4.1 Wait states generator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity WaitStatesGenerator is
    Generic(
        WIDTH : positive    := 17);
    Port(
        WAITSTATES  : in  STD_LOGIC_VECTOR (WIDTH-1 downto 0);
        ENABLE      : in  STD_LOGIC;
        BUSY            : out STD_LOGIC;
        CLK         : in  STD_LOGIC;
        RESET           : in  STD_LOGIC);
end WaitStatesGenerator;

architecture Behavioral of WaitStatesGenerator is
    signal sClkIn           : std_logic;
    attribute buffer_type: string;
    attribute buffer_type of sClkIn: signal is "bufgp";

    signal regWaitStates    : std_logic_vector(WIDTH-1 downto 0);
begin

    sClkIn  <= CLK;
    process(sClkIn, RESET) begin
        if(RESET='1') then
            regWaitStates   <= (others=>'0');
        elsif(rising_edge(sClkIn)) then
            if(regWaitStates>0) then
                regWaitStates   <= regWaitStates - '1';
            elsif(ENABLE='1') then
                regWaitStates   <= WAITSTATES;
            end if;
        end if;
    end process;

    BUSY    <= '0' when (regWaitStates="00000000000000000") else '1';
end Behavioral;
```

4.2 Bus Arbitrer

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RfArb is
    Port(
        OP          : in  STD_LOGIC_VECTOR(7 downto 0);
        PLDSTS   : in  STD_LOGIC_VECTOR(11 downto 0);
        RXIRQ       : in  STD_LOGIC;
        TXIRQ    : in  STD_LOGIC;
        RXEN        : out STD_LOGIC;
        TXEN        : out STD_LOGIC;
        STEN        : out STD_LOGIC;
        CFEN        : out STD_LOGIC;
        RXBUSY   : in  STD_LOGIC;
        TXBUSY   : in  STD_LOGIC;
        STBUSY   : in  STD_LOGIC;
        CFBUSY   : in  STD_LOGIC;
        BUSY        : out STD_LOGIC;
        ENABLE   : in  STD_LOGIC;
        CLK      : in  STD_LOGIC;
      RESET      : in  STD_LOGIC);
end RfArb;

architecture Behavioral of RfArb is
    type SM_States is (S_POR, S_IDLE, S_ST,
                           S_TX, S_RX, S_CFG,
                           S_EXIT);

    signal state    : SM_States := S_POR;
    signal PldRxNew : std_logic_vector(5 downto 0);
    signal PldTxNew : std_logic_vector(5 downto 0);
    signal sRxEn        : std_logic := '0';
    signal sTxEn        : std_logic := '0';
    signal sStEn        : std_logic := '0';
    signal sCfEn        : std_logic := '0';
    signal sWait        : std_logic := '0';
begin
    PldRxNew    <= PLDSTS(11 downto 6);
    PldTxNew    <= PLDSTS(5 downto 0);

    process(CLK, RESET) begin
      if(RESET='1') then
          sRxEn   <= '0';
          sTxEn   <= '0';
          sStEn   <= '0';
          sWait   <= '0';
          state   <= S_POR;
      elsif(rising_edge(CLK)) then
          case (state) is
              when S_POR =>
                  sRxEn   <= '0';
                  sTxEn   <= '0';
                  sStEn   <= '0';
                  sCfEn   <= '0';
                  sWait   <= '0';
                  if((RXBUSY='0') and (TXBUSY='0') and (STBUSY='0')) then
                      state   <= S_ST;
                  end if;
              when S_ST =>
                  if((sStEn='0') and (STBUSY='0') and (sWait='0')) then
```

```
            sStEn   <= '1';
            sWait   <= '1';
        elsif((sStEn='1') and (STBUSY='1') and (sWait='1')) then
            sStEn   <= '0';
        elsif((sStEn='0') and (STBUSY='0') and (sWait='1')) then
            sWait   <= '0';
            state   <= S_IDLE;
        end if;
    when S_IDLE =>
        sRxEn   <= '0';
        sTxEn   <= '0';
        sStEn   <= '0';
        sCfEn   <= '0';
        sWait   <= '0';
        if(ENABLE='1') then
            if(OP=x"00") then
                state   <= S_TX;
            elsif((OP=x"01") or (OP=x"02")) then
                state <= S_CFG;
            end if;
        end if;
    when S_TX =>
        if((sTxEn='0') and (TXBUSY='0') and (sWait='0')) then
            sWait   <= '1';
            sTxEn   <= '1';
        elsif((sTxEn='1') and (TXBUSY='1') and (sWait='1')) then
            sTxEn   <= '0';
        elsif((sTxEn='0') and (TXBUSY='0') and (sWait='1')) then
            sWait   <= '0';
            if(ENABLE='1') then
                if(PLDSTS(11 downto 6) /= "111111") then
                    state   <= S_RX;
                end if;
            else
                state   <= S_EXIT;
            end if;
        end if;
    when S_RX =>
        if((sRxEn='0') and (RXBUSY='0') and (sWait='0')) then
            sWait   <= '1';
            sRxEn   <= '1';
        elsif((sRxEn='1') and (RXBUSY='1') and (sWait='1')) then
            sRxEn   <= '0';
        elsif((sRxEn='0') and (RXBUSY='0') and (sWait='1')) then
            sWait   <= '0';
            if(ENABLE='1') then
                if(PLDSTS(5 downto 0) /= "111111") then
                    state   <= S_TX;
                end if;
            else
                state   <= S_EXIT;
            end if;
        end if;
    when S_CFG =>
        if((sCfEn='0') and (CFBUSY='0') and (sWait='0')) then
            sWait   <= '1';
            sCfEn   <= '1';
        elsif((sCfEn='1') and (CFBUSY='1') and (sWait='1')) then
            sCfEn   <= '0';
        elsif((sCfEn='0') and (CFBUSY='0') and (sWait='1')) then
            sWait   <= '0';
            state   <= S_EXIT;
        end if;
    when S_EXIT =>
```

```
                    sRxEn   <= '0';
                    sTxEn   <= '0';
                    sStEn   <= '0';
                    sCfEn   <= '0';
                    sWait   <= '0';
                    if(ENABLE='0') then
                        state   <= S_IDLE;
                    end if;
                when others =>
                    sRxEn   <= '0';
                    sTxEn   <= '0';
                    sStEn   <= '0';
                    sCfEn   <= '0';
                    sWait   <= '0';
                    if((RXBUSY='0') and (TXBUSY='0') and (STBUSY='0')) then
                        state   <= S_POR;
                    end if;
            end case;
        end if;
    end process;

    RXEN    <= sRxEn;
    TXEN    <= sTxEn;
    STEN    <= sStEn;
    CFEN    <= sCfEn;

    BUSY    <= '0' when (state=S_IDLE) else '1';

end Behavioral;
```

4.3 Bus Pull

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RfPull is
    Port(
        COMMAND : out STD_LOGIC_VECTOR(7 downto 0);
        PLDLEN  : out STD_LOGIC_VECTOR(4 downto 0);
        PLDEN       : out STD_LOGIC;
        PLDWR       : out STD_LOGIC;
        DOUTPLD : out STD_LOGIC_VECTOR(7 downto 0);
        DOUTADD : out STD_LOGIC_VECTOR(4 downto 0);
        DOUTWEN : out STD_LOGIC;
        DINADD  : out STD_LOGIC_VECTOR(4 downto 0);
        CMDEN       : out STD_LOGIC;
        RXCS        : out STD_LOGIC;
        TXCS        : out STD_LOGIC;
        TXCE        : out STD_LOGIC;
        RXCE        : out STD_LOGIC;
        PLDSET  : out STD_LOGIC_VECTOR(11 downto 0);
        DOUT        : out STD_LOGIC_VECTOR(7 downto 0);
        ADD     : out STD_LOGIC_VECTOR(9 downto 0);
        WEN     : out STD_LOGIC;
        BUSY        : in  STD_LOGIC_VECTOR(3 downto 0));
end RfPull;

architecture Behavioral of RfPull is
begin

    COMMAND <= (others=>'0')    when (BUSY="0000") else (others=>'Z');
    PLDLEN  <= (others=>'0')    when (BUSY="0000") else (others=>'Z');
    PLDEN       <= '0'              when (BUSY="0000") else 'Z';
    PLDWR       <= '0'              when (BUSY="0000") else 'Z';
    DOUTPLD <= (others=>'0')    when (BUSY="0000") else (others=>'Z');
    DOUTADD <= (others=>'0')    when (BUSY="0000") else (others=>'Z');
    DOUTWEN <= '0'              when (BUSY="0000") else 'Z';
    DINADD  <= (others=>'0')    when (BUSY="0000") else (others=>'Z');
    CMDEN       <= '0'              when (BUSY="0000") else 'Z';
    RXCS        <= '0'              when (BUSY="0000") else 'Z';
    TXCS        <= '0'              when (BUSY="0000") else 'Z';
    TXCE        <= '0'              when (BUSY="0000") else 'Z';
    RXCE        <= '0'              when (BUSY="0000") else 'Z';
    PLDSET  <= (others=>'0')    when (BUSY="0000") else (others=>'Z');
    DOUT        <= (others=>'0')    when (BUSY="0000") else (others=>'Z');
    ADD     <= (others=>'0')    when (BUSY="0000") else (others=>'Z');
    WEN     <= '0'              when (BUSY="0000") else 'Z';
end Behavioral;
```

4.4 Tx handler

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RfTx is
    Port(
        STATUS  : in  STD_LOGIC_VECTOR(7 downto 0);
        COMMAND : out STD_LOGIC_VECTOR(7 downto 0);
        PLDLEN  : out STD_LOGIC_VECTOR(4 downto 0);
        PLDEN       : out STD_LOGIC;
        PLDWR       : out STD_LOGIC;
        DOUTPLD : out STD_LOGIC_VECTOR(7 downto 0);
        DOUTADD : out STD_LOGIC_VECTOR(4 downto 0);
        DOUTWEN : out STD_LOGIC;
        DINPLD  : in  STD_LOGIC_VECTOR(7 downto 0);
        DINADD  : out STD_LOGIC_VECTOR(4 downto 0);
        CMDEN       : out STD_LOGIC;
        CMDBUSY : in  STD_LOGIC;
        RXCS        : out STD_LOGIC;
        TXCS        : out STD_LOGIC;
        TXCE        : out STD_LOGIC;
        RXCE        : out STD_LOGIC;
        IRQRX       : in  STD_LOGIC;
        IRQTX       : in  STD_LOGIC;
        PLDSET  : out STD_LOGIC_VECTOR(11 downto 0);
        PLDSTS  : in  STD_LOGIC_VECTOR(11 downto 0);
        OP          : in  STD_LOGIC_VECTOR(7 downto 0);
        DIN     : in  STD_LOGIC_VECTOR(7 downto 0);
        DOUT        : out STD_LOGIC_VECTOR(7 downto 0);
        ADD     : out STD_LOGIC_VECTOR(9 downto 0);
        WEN     : out STD_LOGIC;
        EN          : in  STD_LOGIC;
        BUSY        : out STD_LOGIC;
        CLK     : in  STD_LOGIC;
      RESET     : in  STD_LOGIC);
end RfTx;

architecture Behavioral of RfTx is
    type SM_States is (S_IDLE,
                       S_POLL_PIPE,
                       S_READ_STATUS,
                       S_LOAD_ADD, S_SEND_ADD,
                       S_LOAD_PLD_SZ, S_LOAD_PLD, S_SEND_PLD,
                       S_POLL_STATUS, S_CLEAR_IRQ, S_SET_STS, S_INC_PIPE,
                       S_EXIT);

    signal state    : SM_States := S_IDLE;
    signal regCOMMAND   : std_logic_vector(7 downto 0);
    signal regPLDLEN    : std_logic_vector(4 downto 0);
    signal regPLDEN : std_logic;
    signal regPLDWR : std_logic;
    signal regDOUTPLD   : std_logic_vector(7 downto 0);
    signal regDOUTADD   : std_logic_vector(4 downto 0);
    signal regDOUTWEN   : std_logic;
    signal regDINADD    : std_logic_vector(4 downto 0);
    signal regCMDEN : std_logic := '0';
    signal regRXCS      : std_logic := '0';
    signal regTXCS      : std_logic := '0';
    signal regTXCE      : std_logic := '0';
    signal regRXCE      : std_logic := '0';
    signal regPLDSET    : std_logic_vector(11 downto 0);
```

```
    signal regDOUT      : std_logic_vector(7 downto 0);
    signal regADD       : std_logic_vector(9 downto 0);
    signal regWEN       : std_logic;
    signal regBUSY      : std_logic;
    signal sWait    : std_logic := '0';
    signal regSts   : std_logic;
    signal regPipe  : std_logic_vector(2 downto 0) := (others=>'0');
    signal regBase : std_logic_vector(4 downto 0);
    signal regSzAd  : std_logic_vector(9 downto 0);
    signal regPlSz  : std_logic_vector(4 downto 0);
    signal regStat  : std_logic_vector(7 downto 0);
    signal sIrqSts  : std_logic;
    signal sRamWait : std_logic := '1';
begin

    regSts  <= PLDSTS(conv_integer(regPipe));
    regBase <= "00000" when (regPipe="000") else
                   "00001" when (regPipe="001") else
                   "00010" when (regPipe="010") else
                   "00011" when (regPipe="011") else
                   "00100" when (regPipe="100") else
                   "00101" when (regPipe="101") else
                   "00111";

    regSzAd <= "0011000000" when (regPipe="000") else
                   "0011000001" when (regPipe="001") else
                   "0011000010" when (regPipe="010") else
                   "0011000011" when (regPipe="011") else
                   "0011000100" when (regPipe="100") else
                   "0011000101" when (regPipe="101") else
                   "0011111111";

    process(CLK, RESET) begin
        if(RESET='1') then
            regCMDEN        <= '0';
            regRXCS     <= '0';
            regRXCE     <= '0';
            regTXCS     <= '0';
            regTXCE     <= '0';
            regDOUTWEN  <= '0';
            regCMDEN        <= '0';
            regWEN      <= '0';
            sWait           <= '0';
            regPipe     <= (others=>'0');
            regPLDSET   <= (others=>'0');
            sRamWait        <= '1';
            state           <= S_IDLE;
        elsif(rising_edge(CLK)) then
            case (state) is
                when S_IDLE =>
                    regCMDEN        <= '0';
                    regRXCS     <= '0';
                    regRXCE     <= '0';
                    regTXCS     <= '0';
                    regTXCE     <= '0';
                    regDOUTWEN  <= '0';
                    regCMDEN        <= '0';
                    regWEN      <= '0';
                    sWait           <= '0';
                    regPLDSET   <= (others=>'0');
                    regPLDSET   <= (others=>'0');
                    if(EN='1') then
                        state       <= S_POLL_PIPE;
                    end if;
```

```
when S_POLL_PIPE =>
    regPLDSET   <= (others=>'0');
    if(EN='0') then
        state       <= S_EXIT;
    elsif(regSts='0')then
        regDOUTADD  <= (others=>'0');
        if(regPipe="000") then
            regADD      <= "1000000110";
        else
            regADD      <= "1000001011";
        end if;
        sWait           <= '0';
        sRamWait        <= '0';
        regCOMMAND  <= x"FF";
        regPLDEN        <= '0';
        regPLDWR        <= '0';
        sWait           <= '0';
        state           <= S_READ_STATUS;
    else
        if(regPipe="101") then
            regPipe <= (others=>'0');
        else
            regPipe <= regPipe + '1';
        end if;
    end if;
when S_READ_STATUS =>
    if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
        sIrqSts <= IRQTX;
        regTXCS <= '1';
        regCMDEN    <= '1';
    elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
        regCMDEN    <= '0';
        sWait       <= '1';
    elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
        regTXCS     <= '0';
        sWait       <= '0';
        if(sIrqSts=IRQTX)then
            regStat <= STATUS;
            if(STATUS(6 downto 4)="000")then
                state   <= S_LOAD_ADD;
            else
                regCOMMAND  <= x"FF";
                regPLDEN        <= '0';
                regPLDWR        <= '0';
                state           <= S_POLL_STATUS;
            end if;
        end if;
    end if;
when S_LOAD_ADD =>
    if(sRamWait='0') then
        regDOUTPLD  <= DIN;
        if(regDOUTADD="00011") then
            case (regPipe) is
                when "000" =>
                    regADD  <= regADD + '1';
                when "001" =>
                    regADD  <= regADD + '1';
                when "010" =>
                    regADD  <= "1000010000";
                when "011" =>
                    regADD  <= "1000010001";
                when "100" =>
                    regADD  <= "1000010010";
                when "101" =>
```

```
                    regADD  <= "1000010011";
                when others =>
                    regADD  <= regADD + '1';
            end case;
        else
            regADD  <= regADD + '1';
        end if;
        if(regDOUTWEN='1') then
            regDOUTADD  <= regDOUTADD + '1';
        else
            regDOUTWEN  <= '1';
        end if;
        if(regDOUTADD="00100") then
            sWait       <= '0';
            regCOMMAND  <= x"30";
            regPLDEN        <= '1';
            regPLDLEN   <= "00100";
            regPLDWR        <= '1';
            state           <= S_SEND_ADD;
        end if;
    else
        sRamWait    <= '0';
    end if;
when S_SEND_ADD =>
    regDOUTWEN  <= '0';
    if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
        regTXCS <= '1';
        regCMDEN    <= '1';
    elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
        regCMDEN    <= '0';
        sWait       <= '1';
    elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
        regTXCS     <= '0';
        sWait       <= '0';
        if(regCOMMAND=x"30")then
            regCOMMAND  <= x"2A";
        else
            regADD      <= regSzAd;
            state           <= S_LOAD_PLD_SZ;
            sRamWait        <= '1';
        end if;
    end if;
when S_LOAD_PLD_SZ =>
    if(sRamWait='0')then
        if(DIN=x"00") then
            regPLDSET(conv_integer(regPipe))    <= '1';
            state   <= S_INC_PIPE;
        else
            regPLDLEN<= DIN - '1';
            regPLDEN    <= '1';
            sWait       <= '0';
            regADD  <= regBase & "00000";
            regDOUTADD  <= (others=>'0');
            state       <= S_LOAD_PLD;
            sRamWait    <= '1';
        end if;
    else
        sRamWait    <= '0';
    end if;
when S_LOAD_PLD =>
    if(sRamWait='0')then
        if(regDOUTWEN='1') then
            regDOUTADD  <= regDOUTADD + '1';
        end if;
```

```
            if(regDOUTADD(4 downto 0)=regPLDLEN) then
                sWait           <= '0';
                regCOMMAND  <= x"A0";
                regPLDEN        <= '1';
                regPLDWR        <= '1';
                regDOUTWEN  <= '0';
                state           <= S_SEND_PLD;
            else
                regDOUTPLD  <= DIN;
                regADD      <= regADD + '1';
                regDOUTWEN  <= '1';
            end if;
        else
            regADD  <= regADD + '1';
            sRamWait    <= '0';
        end if;
    when S_SEND_PLD =>
        regDOUTWEN  <= '0';
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regTXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regTXCS     <= '0';
            sWait           <= '0';
            regCOMMAND  <= x"FF";
            regPLDEN        <= '0';
            regPLDWR        <= '0';
            state           <= S_POLL_STATUS;
        end if;
    when S_POLL_STATUS =>
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            sIrqSts <= IRQTX;
            regTXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regTXCS     <= '0';
            sWait           <= '0';
            if((STATUS(5 downto 4)/="00") and (sIrqSts=IRQTX)) then
                regStat     <= STATUS;
                regCOMMAND  <= x"27";
                regPLDEN        <= '1';
                regPLDLEN   <= (others=>'0');
                regPLDWR        <= '0';
                regDINADD   <= (others=>'0');
                regDOUTADD  <= (others=>'0');
                regDOUTPLD  <= '0' & STATUS(6 downto 4) & "0000";
                regDOUTWEN  <= '1';
                state       <= S_CLEAR_IRQ;
            end if;
        end if;
    when S_CLEAR_IRQ =>
        regDOUTWEN  <= '0';
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regTXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
```

```
                elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
                    regTXCS     <= '0';
                    sWait           <= '0';
                    if(STATUS(6 downto 4) = "000")then
                        if(regStat(5)='1')then
                            state   <= S_SET_STS;
                        else--if(regStat(4)='1')then
                            state   <= S_INC_PIPE;
                        end if;
                    end if;
                end if;
            when S_SET_STS =>
                regPLDSET(conv_integer(regPipe))    <= '1';
                state   <= S_INC_PIPE;
            when S_INC_PIPE =>
                regPLDSET   <= (others=>'0');
                if(regPipe="101") then
                    regPipe <= (others=>'0');
                else
                    regPipe <= regPipe + '1';
                end if;
                state   <= S_POLL_PIPE;
            when S_EXIT =>
                if(EN='0') then
                    state   <= S_IDLE;
                end if;
            when others =>
                regCMDEN        <= '0';
                regRXCS     <= '0';
                regRXCE     <= '0';
                regTXCS     <= '0';
                regTXCE     <= '0';
                regDOUTWEN  <= '0';
                regCMDEN        <= '0';
                regWEN      <= '0';
                sWait           <= '0';
                regPipe     <= (others=>'0');
                regPLDSET   <= (others=>'0');
                sRamWait        <= '1';
                state           <= S_IDLE;
        end case;
    end if;
end process;

regBUSY <= '0' when ((state=S_IDLE) or (state=S_EXIT)) else '1';

COMMAND <= regCOMMAND   when (regBUSY='1') else (others=>'Z');
PLDLEN  <= regPLDLEN    when (regBUSY='1') else (others=>'Z');
PLDEN       <= regPLDEN     when (regBUSY='1') else 'Z';
PLDWR       <= regPLDWR     when (regBUSY='1') else 'Z';
DOUTPLD <= regDOUTPLD   when (regBUSY='1') else (others=>'Z');
DOUTADD <= regDOUTADD   when (regBUSY='1') else (others=>'Z');
DOUTWEN <= regDOUTWEN   when (regBUSY='1') else 'Z';
DINADD  <= regDINADD    when (regBUSY='1') else (others=>'Z');
CMDEN       <= regCMDEN     when (regBUSY='1') else 'Z';
RXCS        <= regRXCS      when (regBUSY='1') else 'Z';
TXCS        <= regTXCS      when (regBUSY='1') else 'Z';
TXCE        <= regTXCE      when (regBUSY='1') else 'Z';
RXCE        <= regRXCE      when (regBUSY='1') else 'Z';
PLDSET  <= regPLDSET    when (regBUSY='1') else (others=>'Z');
DOUT        <= regDOUT      when (regBUSY='1') else (others=>'Z');
ADD     <= regADD       when (regBUSY='1') else (others=>'Z');
WEN     <= regWEN       when (regBUSY='1') else 'Z';
BUSY        <= regBUSY;
```

```
end Behavioral;
```

4.5 Rx handler

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RfRx is
    Port(
        STATUS  : in  STD_LOGIC_VECTOR(7 downto 0);
        COMMAND : out STD_LOGIC_VECTOR(7 downto 0);
        PLDLEN  : out STD_LOGIC_VECTOR(4 downto 0);
        PLDEN       : out STD_LOGIC;
        PLDWR       : out STD_LOGIC;
        DOUTPLD : out STD_LOGIC_VECTOR(7 downto 0);
        DOUTADD : out STD_LOGIC_VECTOR(4 downto 0);
        DOUTWEN : out STD_LOGIC;
        DINPLD  : in  STD_LOGIC_VECTOR(7 downto 0);
        DINADD  : out STD_LOGIC_VECTOR(4 downto 0);
        CMDEN       : out STD_LOGIC;
        CMDBUSY : in  STD_LOGIC;
        RXCS        : out STD_LOGIC;
        TXCS        : out STD_LOGIC;
        TXCE        : out STD_LOGIC;
        RXCE        : out STD_LOGIC;
        IRQRX       : in  STD_LOGIC;
        IRQTX       : in  STD_LOGIC;
        PLDSET  : out STD_LOGIC_VECTOR(11 downto 0);
        PLDSTS  : in  STD_LOGIC_VECTOR(11 downto 0);
        OP          : in  STD_LOGIC_VECTOR(7 downto 0);
        DIN     : in  STD_LOGIC_VECTOR(7 downto 0);
        DOUT        : out STD_LOGIC_VECTOR(7 downto 0);
        ADD     : out STD_LOGIC_VECTOR(9 downto 0);
        WEN     : out STD_LOGIC;
        EN          : in  STD_LOGIC;
        BUSY        : out STD_LOGIC;
        CLK     : in  STD_LOGIC;
      RESET     : in  STD_LOGIC);
end RfRx;

architecture Behavioral of RfRx is
    type SM_States is (S_IDLE,
                       S_FIFO_STATUS, S_SAVE_PIPE, S_WID, S_READ_PLD, S_SAVE_PLD,
S_SAVE_WID,
                       S_SET_RX_FLAG, S_WAIT_CE_DEASSERT, S_CLEAR_DR_FLAG,
S_WAIT_CE_ASSERT,
                       S_EXIT);

    signal state    : SM_States := S_IDLE;

    COMPONENT WaitStatesGenerator
    PORT(
        WAITSTATES : IN std_logic_vector(16 downto 0);
        ENABLE : IN std_logic;
        BUSY : OUT std_logic;
        CLK : IN std_logic;
        RESET : IN std_logic
        );
    END COMPONENT;

    signal  regWaitStates   : std_logic_vector(16 downto 0);
    signal  regWaitEn   : std_logic;
    signal  regWaitBusy     : std_logic;
    signal regCOMMAND   : std_logic_vector(7 downto 0);
```

```
    signal regPLDLEN    : std_logic_vector(4 downto 0);
    signal regPLDEN : std_logic;
    signal regPLDWR : std_logic;
    signal regDOUTPLD   : std_logic_vector(7 downto 0);
    signal regDOUTADD   : std_logic_vector(4 downto 0);
    signal regDOUTWEN   : std_logic;
    signal regDINADD    : std_logic_vector(4 downto 0);
    signal regCMDEN : std_logic := '0';
    signal regRXCS      : std_logic := '0';
    signal regTXCS      : std_logic := '0';
    signal regTXCE      : std_logic := '0';
    signal regRXCE      : std_logic := '0';
    signal regPLDSET    : std_logic_vector(11 downto 0);
    signal regDOUT      : std_logic_vector(7 downto 0);
    signal regADD       : std_logic_vector(9 downto 0);
    signal regWEN       : std_logic;
    signal regBUSY      : std_logic;
    signal sWait    : std_logic := '0';
    signal regSts   : std_logic;
    signal regPipe  : std_logic_vector(2 downto 0);
    signal regWID   : std_logic_vector(7 downto 0);
    signal regWID2  : std_logic_vector(7 downto 0);
    signal regCount: std_logic_vector(7 downto 0);
    signal regBase  : std_logic_vector(4 downto 0);
    signal regIrqRx : std_logic;
begin
    UWS: WaitStatesGenerator PORT MAP(
        WAITSTATES => regWaitStates,
        ENABLE => regWaitEn,
        BUSY => regWaitBusy,
        CLK => CLK,
        RESET => RESET
    );

    regSts  <= PLDSTS(conv_integer(regPipe) + 6);

    regBase <= "01000" when (regPipe="000") else
               "01001" when (regPipe="001") else
               "01010" when (regPipe="010") else
               "01011" when (regPipe="011") else
               "01100" when (regPipe="100") else
               "01101" when (regPipe="101") else
               "01111";

    process(CLK, RESET) begin
      if(RESET='1') then
          regCMDEN          <= '0';
          regRXCS           <= '0';
          regRXCE           <= '0';
          regTXCS           <= '0';
          regTXCE           <= '0';
          sWait             <= '0';
          regWaitEn         <= '0';
          state             <= S_IDLE;
      elsif(rising_edge(CLK)) then
          case (state) is
              when S_IDLE =>
                  regCMDEN    <= '0';
                  regPLDSET   <= (others=>'0');
                  if(EN='1') then
                      sWait         <= '0';
                      regCOMMAND  <= x"17";
                      regPLDEN      <= '1';
                      regPLDLEN   <= (others=>'0');
```

```
            regPLDWR        <= '0';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '0';
            state           <= S_FIFO_STATUS;
        end if;
    when S_FIFO_STATUS =>
        regPLDSET   <= (others=>'0');
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regIrqRx    <= IRQRX;
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';
            if(regIrqRx=IRQRX) then
                regPipe <= STATUS(3 downto 1);
                state       <= S_SAVE_PIPE;
            end if;
        end if;
    when S_SAVE_PIPE =>
        if((regSts='0') and (DINPLD(0)='0') and (regPipe/="111")) then
            regCOMMAND  <= x"60";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '0';
            regDINADD   <= (others=>'0');
            state           <= S_WID;
        else
            state   <= S_EXIT;
        end if;
    when S_WID =>
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';

            regWID      <= DINPLD - '1';
            regWID2     <= DINPLD;
            regCount        <= (others=>'0');
            regCOMMAND  <= x"61";
            regPLDEN        <= '1';
            regPLDWR        <= '0';
            regDINADD   <= (others=>'0');
            state           <= S_READ_PLD;
        end if;
    when S_READ_PLD =>
        regPLDLEN   <= regWID(4 downto 0);
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS     <= '0';
```

```
            sWait           <= '0';

            regDINADD   <= (others=>'0');
            regADD      <= regBase & "00000";
            state           <= S_SAVE_PLD;
        end if;
    when S_SAVE_PLD =>
        regDOUT     <= DINPLD;
        regDINADD   <= regDINADD + '1';
        if(regWEN='0') then
            regWEN  <= '1';
        else
            regADD  <= regADD + '1';
            regCount        <= regCount + '1';
            if(regWID=regCount) then
                regWEN  <= '0';
                state       <= S_SAVE_WID;
            else
            end if;
        end if;
    when S_SAVE_WID =>
        regDOUT <= regWID2;
        regADD  <= "0111000" & regPipe;
        if(regWEN='0') then
            regWEN  <= '1';
        else
            regWEN  <= '0';
            state       <= S_SET_RX_FLAG;
        end if;
    when S_SET_RX_FLAG =>
        regPLDSET(conv_integer(regPipe)+6)  <= '1';
        sWait               <= '0';
        regRXCE             <= '1';
        regWaitStates   <= conv_std_logic_vector(50, 17);
        state               <= S_WAIT_CE_DEASSERT;
    when S_WAIT_CE_DEASSERT =>
        regPLDSET   <= (others=>'0');
        if((regWaitEn='0') and (regWaitBusy='0') and (sWait='0')) then
            regWaitEn   <= '1';
        elsif((regWaitEn='1') and (regWaitBusy='1') and (sWait='0')) then
            regWaitEn   <= '0';
            sWait           <= '1';
        elsif((regWaitEn='0') and (regWaitBusy='0') and (sWait='1')) then
            sWait               <= '0';
            regCOMMAND      <= x"27";
            regPLDEN            <= '1';
            regPLDLEN       <=  (others=>'0');
            regPLDWR            <= '1';
            regDINADD       <= (others=>'0');
            regDOUTADD      <= (others=>'0');
            regDOUTPLD      <= x"40";
            regDOUTWEN      <= '1';
            state               <= S_CLEAR_DR_FLAG;
        end if;
    when S_CLEAR_DR_FLAG =>
        regDOUTWEN  <= '0';
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS     <= '0';
```

```
                    sWait           <= '0';
                    regRXCE     <= '0';
                    regWaitStates   <= conv_std_logic_vector(50, 17);
                    state           <= S_WAIT_CE_ASSERT;
                end if;
            when S_WAIT_CE_ASSERT =>
                if((regWaitEn='0') and (regWaitBusy='0') and (sWait='0')) then
                    regWaitEn   <= '1';
                elsif((regWaitEn='1') and (regWaitBusy='1') and (sWait='0')) then
                    regWaitEn   <= '0';
                    sWait           <= '1';
                elsif((regWaitEn='0') and (regWaitBusy='0') and (sWait='1')) then
                    if(EN='1') then
                        sWait           <= '0';
                        regCOMMAND  <= x"17";
                        regPLDEN        <= '1';
                        regPLDLEN   <= (others=>'0');
                        regPLDWR        <= '0';
                        regDINADD   <= (others=>'0');
                        regDOUTADD  <= (others=>'0');
                        regDOUTWEN  <= '0';
                        state           <= S_FIFO_STATUS;
                    else
                        state   <= S_EXIT;
                    end if;
                end if;
            when S_EXIT =>
                if(EN='0') then
                    state   <= S_IDLE;
                end if;
            when others =>
                regCMDEN        <= '0';
                regRXCS         <= '0';
                regRXCE         <= '0';
                regTXCS         <= '0';
                regTXCE         <= '0';
                sWait           <= '0';
                regWaitEn       <= '0';
                state           <= S_IDLE;
        end case;
    end if;
end process;

regBUSY <= '0' when ((state=S_IDLE) or (state=S_EXIT)) else '1';

COMMAND <= regCOMMAND   when (regBUSY='1') else (others=>'Z');
PLDLEN  <= regPLDLEN    when (regBUSY='1') else (others=>'Z');
PLDEN       <= regPLDEN     when (regBUSY='1') else 'Z';
PLDWR       <= regPLDWR     when (regBUSY='1') else 'Z';
DOUTPLD <= regDOUTPLD   when (regBUSY='1') else (others=>'Z');
DOUTADD <= regDOUTADD   when (regBUSY='1') else (others=>'Z');
DOUTWEN <= regDOUTWEN   when (regBUSY='1') else 'Z';
DINADD  <= regDINADD    when (regBUSY='1') else (others=>'Z');
CMDEN       <= regCMDEN     when (regBUSY='1') else 'Z';
RXCS        <= regRXCS      when (regBUSY='1') else 'Z';
TXCS        <= regTXCS      when (regBUSY='1') else 'Z';
TXCE        <= regTXCE      when (regBUSY='1') else 'Z';
RXCE        <= regRXCE      when (regBUSY='1') else 'Z';
PLDSET  <= regPLDSET    when (regBUSY='1') else (others=>'Z');
DOUT        <= regDOUT      when (regBUSY='1') else (others=>'Z');
ADD     <= regADD       when (regBUSY='1') else (others=>'Z');
WEN     <= regWEN       when (regBUSY='1') else 'Z';

BUSY        <= regBUSY;
```

```
end Behavioral;
```

4.6 Initialization

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RfSt is
    Port(
        STATUS  : in  STD_LOGIC_VECTOR(7 downto 0);
        COMMAND : out STD_LOGIC_VECTOR(7 downto 0);
        PLDLEN  : out STD_LOGIC_VECTOR(4 downto 0);
        PLDEN       : out STD_LOGIC;
        PLDWR       : out STD_LOGIC;
        DOUTPLD : out STD_LOGIC_VECTOR(7 downto 0);
        DOUTADD : out STD_LOGIC_VECTOR(4 downto 0);
        DOUTWEN : out STD_LOGIC;
        DINPLD  : in  STD_LOGIC_VECTOR(7 downto 0);
        DINADD  : out STD_LOGIC_VECTOR(4 downto 0);
        CMDEN       : out STD_LOGIC;
        CMDBUSY : in  STD_LOGIC;
        RXCS        : out STD_LOGIC;
        TXCS        : out STD_LOGIC;
        TXCE        : out STD_LOGIC;
        RXCE        : out STD_LOGIC;
        IRQRX       : in  STD_LOGIC;
        IRQTX       : in  STD_LOGIC;
        PLDSET  : out STD_LOGIC_VECTOR(11 downto 0);
        PLDSTS  : in  STD_LOGIC_VECTOR(11 downto 0);
        OP          : in  STD_LOGIC_VECTOR(7 downto 0);
        DIN     : in  STD_LOGIC_VECTOR(7 downto 0);
        DOUT        : out STD_LOGIC_VECTOR(7 downto 0);
        ADD     : out STD_LOGIC_VECTOR(9 downto 0);
        WEN     : out STD_LOGIC;
        EN          : in  STD_LOGIC;
        BUSY        : out STD_LOGIC;
        CFG_SEL : in  STD_LOGIC;
        CLK     : in  STD_LOGIC;
      RESET     : in  STD_LOGIC);
end RfSt;

architecture Behavioral of RfSt is
    COMPONENT ROM_128x10
    PORT (
        a       : IN STD_LOGIC_VECTOR(6 downto 0);
        spo : OUT STD_LOGIC_VECTOR(9 downto 0));
    END COMPONENT;

    COMPONENT ROM_128x10_Init0
    PORT(
        ADD : IN std_logic_vector(6 downto 0);
        DOUT : OUT std_logic_vector(9 downto 0)
        );
    END COMPONENT;

    COMPONENT ROM_128x10_Init1
    PORT(
        ADD : IN std_logic_vector(6 downto 0);
        DOUT : OUT std_logic_vector(9 downto 0)
        );
    END COMPONENT;

    signal  RomAdd  : std_logic_vector(6 downto 0);
    signal  RomDout : std_logic_vector(9 downto 0);
```

```
    signal  RomAdd0 : std_logic_vector(6 downto 0);
    signal  RomDout0    : std_logic_vector(9 downto 0);
    signal  RomAdd1 : std_logic_vector(6 downto 0);
    signal  RomDout1    : std_logic_vector(9 downto 0);

    COMPONENT WaitStatesGenerator
    PORT(
        WAITSTATES : IN std_logic_vector(16 downto 0);
        ENABLE : IN std_logic;
        BUSY : OUT std_logic;
        CLK : IN std_logic;
        RESET : IN std_logic
        );
    END COMPONENT;

    signal regWaitStates   : std_logic_vector(16 downto 0);
    signal regWaitEnable   : std_logic;
    signal regWaitBusy     : std_logic;
    signal regCOMMAND   : std_logic_vector(7 downto 0);
    signal regPLDLEN    : std_logic_vector(4 downto 0);
    signal regPLDEN : std_logic;
    signal regPLDWR : std_logic;
    signal regDOUTPLD   : std_logic_vector(7 downto 0);
    signal regDOUTADD   : std_logic_vector(4 downto 0);
    signal regDOUTWEN   : std_logic;
    signal regDINADD    : std_logic_vector(4 downto 0);
    signal regCMDEN : std_logic := '0';
    signal regRXCS      : std_logic := '0';
    signal regTXCS      : std_logic := '0';
    signal regTXCE      : std_logic := '0';
    signal regRXCE      : std_logic := '0';
    signal regPLDSET    : std_logic_vector(11 downto 0);
    signal regDOUT      : std_logic_vector(7 downto 0);
    signal regADD       : std_logic_vector(9 downto 0);
    signal regWEN       : std_logic;
    signal regBUSY      : std_logic;

    type SM_States is (S_IDLE, S_WAIT_CE_DEASSERT,
                       S_WRITE_FEATURE, S_READ_FEATURE, S_ACTIVATE_FEATURE,
                       S_WRITE_CMD, S_WRITE_PLD, S_EXECUTE,
                       S_EXIT);

    signal state    : SM_States := S_IDLE;
    signal sWait    : std_logic := '0';
    signal sFeatRx  : std_logic := '0';

begin

    RCF0: ROM_128x10_Init0 PORT MAP (
        ADD => RomAdd,
        DOUT => RomDout0
    );

    RCF1: ROM_128x10_Init1 PORT MAP (
        ADD => RomAdd,
        DOUT => RomDout1
    );

    RomDout <= RomDout0 when (CFG_SEL='0') else RomDout1;

    UWS: WaitStatesGenerator PORT MAP(
        WAITSTATES => regWaitStates,
        ENABLE => regWaitEnable,
        BUSY => regWaitBusy,
```

```
    CLK => CLK,
    RESET => RESET
);

process(CLK, RESET) begin
    if(RESET='1') then
        regWaitStates   <= (others=>'0');
        regWaitEnable   <= '0';
        regCMDEN            <= '0';
        regTXCE         <= '0';
        regRXCE         <= '0';
        regTXCS         <= '0';
        regRXCS         <= '0';
        sWait               <= '0';
        RomAdd          <= (others=>'0');
        state               <= S_IDLE;
    elsif(rising_edge(CLK)) then
        case (state) is
            when S_IDLE =>
                if((EN='1') and (CMDBUSY='0')) then
                    regTXCE <= '1';
                    regRXCE <= '1';
                    regTXCS <= '0';
                    regRXCS <= '0';
                    sFeatRx <= '0';
                    regWaitStates   <= conv_std_logic_vector(75000, 17);
                    state               <= S_WAIT_CE_DEASSERT;
                end if;
            when S_WAIT_CE_DEASSERT =>
                if((regWaitEnable='0') and (regWaitBusy='0') and (sWait='0')) then
                    regWaitEnable   <= '1';
                elsif((regWaitEnable='1') and (regWaitBusy='1') and (sWait='0')) then
                    regWaitEnable   <= '0';
                    sWait               <= '1';
                elsif((regWaitEnable='0') and (regWaitBusy='0') and (sWait='1')) then
                    sWait           <= '0';
                    RomAdd          <= (others=>'0');
                    regCOMMAND  <= x"3D";
                    regPLDEN        <= '1';
                    regPLDLEN   <= (others=>'0');
                    regPLDWR        <= '1';
                    regDOUTADD  <= (others=>'0');
                    regDOUTPLD  <= x"07";
                    regDOUTWEN  <= '1';
                    state           <= S_WRITE_FEATURE;
                end if;
            when S_WRITE_FEATURE =>
                regDOUTWEN  <= '0';
                if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
                    if(sFeatRx='0') then
                        regTXCS <= '1';
                    else
                        regRXCS <= '1';
                    end if;
                    regCMDEN    <= '1';
                elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
                    regCMDEN    <= '0';
                    sWait       <= '1';
                elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
                    sWait   <= '0';
                    regTXCS <= '0';
                    regRXCS <= '0';
                    regCOMMAND  <= x"1D";
                    regPLDEN        <= '1';
```

```
                regPLDLEN   <= (others=>'0');
                regPLDWR        <= '0';
                regDINADD   <= (others=>'0');
                state           <= S_READ_FEATURE;
            end if;
        when S_READ_FEATURE =>
            if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
                if(sFeatRx='0') then
                    regTXCS <= '1';
                else
                    regRXCS <= '1';
                end if;
                regCMDEN    <= '1';
            elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
                regCMDEN    <= '0';
                sWait       <= '1';
            elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
                sWait   <= '0';
                regTXCS <= '0';
                regRXCS <= '0';
                if(DINPLD=x"07") then
                    if(sFeatRx='0') then
                        sFeatRx <= '1';
                        state       <= S_WAIT_CE_DEASSERT;
                    else
                        state   <= S_WRITE_CMD;
                    end if;
                else
                    regCOMMAND  <= x"50";
                    regPLDEN        <= '1';
                    regPLDLEN   <= (others=>'0');
                    regPLDWR        <= '1';
                    regDOUTADD  <= (others=>'0');
                    regDOUTPLD  <= x"73";
                    regDOUTWEN  <= '1';
                    state           <= S_ACTIVATE_FEATURE;
                end if;
            end if;
        when S_ACTIVATE_FEATURE =>
            regDOUTWEN  <= '0';
            if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
                if(sFeatRx='0') then
                    regTXCS <= '1';
                else
                    regRXCS <= '1';
                end if;
                regCMDEN    <= '1';
            elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
                regCMDEN    <= '0';
                sWait       <= '1';
            elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
                sWait       <= '0';
                regTXCS <= '0';
                regRXCS <= '0';
                if(sFeatRx='0') then
                    sFeatRx <= '1';
                    state       <= S_WAIT_CE_DEASSERT;
                else
                    state       <= S_WRITE_CMD;
                end if;
            end if;
        when S_WRITE_CMD =>
            if((RomDout(9 downto 8)/="11") and (RomDout(9 downto 8)/="00") and
(RomAdd/="1111111")) then
```

```
            regTXCS     <= RomDout(8);
            regRXCS     <= RomDout(9);
            regCOMMAND  <= RomDout(7 downto 0);
            regPLDEN        <= '0';
            regPLDWR        <= '1';
            regPLDLEN   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '0';
            RomAdd      <= RomAdd + '1';
            state           <= S_WRITE_PLD;
        else
            state   <= S_EXIT;
        end if;
    when S_WRITE_PLD =>
        if(RomDout(9 downto 8)="00") then
            regPLDEN        <= '1';
            regDOUTPLD  <= RomDout(7 downto 0);
            if(regPLDEN='1') then
                regPLDLEN   <= regPLDLEN + '1';
            end if;
            regDOUTADD  <= regPLDLEN;
            if(regDOUTWEN='0') then
                regDOUTWEN  <= '1';
            end if;
            if(RomAdd/="1111111") then
                RomAdd  <= RomAdd + '1';
            else
                state   <= S_EXECUTE;
            end if;
        else
            regDOUTADD  <= regPLDLEN;
            state           <= S_EXECUTE;
        end if;
    when S_EXECUTE =>
        regDOUTWEN  <= '0';
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            sWait       <= '1';
            regCMDEN    <= '0';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            sWait       <= '0';
            regTXCS <= '0';
            regRXCS <= '0';
            if((RomAdd/="1111111") and (RomDout(9 downto 8)/="11")) then
                state   <= S_WRITE_CMD;
            else
                state   <= S_EXIT;
            end if;
        end if;
    when S_EXIT =>
        if(EN='0') then
            regTXCE <= '0';
            regRXCE <= '0';
            regTXCS <= '0';
            regRXCS <= '0';
            RomAdd  <= (others=>'0');
            state       <= S_IDLE;
        end if;
    when others =>
        regWaitStates   <= (others=>'0');
        regWaitEnable   <= '0';
        regCMDEN            <= '0';
        sWait               <= '0';
```

```
                    regTXCE         <= '0';
                    regRXCE         <= '0';
                    regTXCS         <= '0';
                    regRXCS         <= '0';
                    state               <= S_IDLE;
                end case;
            end if;
        end process;

        regBUSY <= '0' when ((state=S_IDLE) or (state=S_EXIT)) else '1';
        COMMAND <= regCOMMAND   when (regBUSY='1') else (others=>'Z');
        PLDLEN  <= regPLDLEN    when (regBUSY='1') else (others=>'Z');
        PLDEN       <= regPLDEN     when (regBUSY='1') else 'Z';
        PLDWR       <= regPLDWR     when (regBUSY='1') else 'Z';
        DOUTPLD <= regDOUTPLD   when (regBUSY='1') else (others=>'Z');
        DOUTADD <= regDOUTADD   when (regBUSY='1') else (others=>'Z');
        DOUTWEN <= regDOUTWEN   when (regBUSY='1') else 'Z';
        DINADD  <= regDINADD    when (regBUSY='1') else (others=>'Z');
        CMDEN       <= regCMDEN     when (regBUSY='1') else 'Z';
        RXCS        <= regRXCS      when (regBUSY='1') else 'Z';
        TXCS        <= regTXCS      when (regBUSY='1') else 'Z';
        TXCE        <= regTXCE      when (regBUSY='1') else 'Z';
        RXCE        <= regRXCE      when (regBUSY='1') else 'Z';
        PLDSET  <= regPLDSET    when (regBUSY='1') else (others=>'Z');
        DOUT        <= regDOUT      when (regBUSY='1') else (others=>'Z');
        ADD     <= regADD       when (regBUSY='1') else (others=>'Z');
        WEN     <= regWEN       when (regBUSY='1') else 'Z';
        BUSY        <= regBUSY;
    end Behavioral;
```

4.7 Initialization ROM (example with TX/RX channels 2/4)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ROM_128x10_Init0 is
    Generic(
        WIDTH   : positive := 10;
        DEPTH : positive := 7);
    Port(
        ADD : in  STD_LOGIC_VECTOR (DEPTH-1 downto 0);
        DOUT    : out  STD_LOGIC_VECTOR (WIDTH-1 downto 0));
end ROM_128x10_Init0;

architecture Behavioral of ROM_128x10_Init0 is
    type rom_type is array (0 to ((2**DEPTH)-1)) of std_logic_vector (WIDTH-1 downto 0);
    constant ROM_NAME : rom_type :=(x"1E1",x"1E2",x"2E1",x"2E2",x"120",x"00A",x"121",x"001",
                                    x"122",x"001",x"123",x"001",x"124",x"01F",
                                    x"125",x"002", -- Tx Channel

x"126",x"00F",x"127",x"070",x"12A",x"0E7",x"0E7",x"0E7",

x"0E7",x"0E7",x"130",x"0E7",x"0E7",x"0E7",x"0E7",x"0E7",

x"131",x"020",x"13C",x"03F",x"13D",x"007",x"220",x"00B",

x"221",x"03F",x"222",x"03F",x"223",x"001",x"224",x"01F",
                                    x"225",x"004", -- Rx Channel
                                    x"226",x"00F",x"227",x"070",x"22A",x"0E7",

x"0E7",x"0E7",x"0E7",x"0E7",x"22B",x"0C2",x"0C2",x"0C2",

x"0C2",x"0C2",x"22C",x"0C3",x"22D",x"0C4",x"22E",x"0C5",

x"22F",x"0C6",x"231",x"020",x"232",x"020",x"233",x"020",

x"234",x"020",x"235",x"020",x"236",x"020",x"23C",x"03F",

x"23D",x"007",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",

x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",

x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",

x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",

x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF",x"3FF");

    signal rom_data : std_logic_vector (WIDTH-1 downto 0);
begin
    DOUT        <= rom_data;
    rom_data    <= ROM_NAME(conv_integer(ADD));
end Behavioral;
```

4.8 Configuration

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RfCfg is
    Port(
        STATUS  : in  STD_LOGIC_VECTOR(7 downto 0);
        COMMAND : out STD_LOGIC_VECTOR(7 downto 0);
        PLDLEN  : out STD_LOGIC_VECTOR(4 downto 0);
        PLDEN       : out STD_LOGIC;
        PLDWR       : out STD_LOGIC;
        DOUTPLD : out STD_LOGIC_VECTOR(7 downto 0);
        DOUTADD : out STD_LOGIC_VECTOR(4 downto 0);
        DOUTWEN : out STD_LOGIC;
        DINPLD  : in  STD_LOGIC_VECTOR(7 downto 0);
        DINADD  : out STD_LOGIC_VECTOR(4 downto 0);
        CMDEN       : out STD_LOGIC;
        CMDBUSY : in  STD_LOGIC;
        RXCS        : out STD_LOGIC;
        TXCS        : out STD_LOGIC;
        TXCE        : out STD_LOGIC;
        RXCE        : out STD_LOGIC;
        IRQRX       : in  STD_LOGIC;
        IRQTX       : in  STD_LOGIC;
        PLDSET  : out STD_LOGIC_VECTOR(11 downto 0);
        PLDSTS  : in  STD_LOGIC_VECTOR(11 downto 0);
        OP          : in  STD_LOGIC_VECTOR(7 downto 0);
        DIN     : in  STD_LOGIC_VECTOR(7 downto 0);
        DOUT        : out STD_LOGIC_VECTOR(7 downto 0);
        ADD     : out STD_LOGIC_VECTOR(9 downto 0);
        WEN     : out STD_LOGIC;
        EN          : in  STD_LOGIC;
        BUSY        : out STD_LOGIC;
        CLK     : in  STD_LOGIC;
      RESET     : in  STD_LOGIC);
end RfCfg;

architecture Behavioral of RfCfg is
    type SM_States is (S_IDLE,
                       S_RD_EN_RXADDR, S_RD_SETUP_AW, S_RD_SETUP_RETR,
                       S_RD_TX_RF_CH, S_RD_RX_RF_CH, S_RD_RF_SETUP,
                       S_RD_ADDR_P0, S_RD_ADDR_P1, S_RD_ADDR_P2,
                       S_RD_ADDR_P3, S_RD_ADDR_P4, S_RD_ADDR_P5, S_STALL,
                       S_WAIT_CE_DEASSERT,
                       S_WR_EN_RXADDR, S_WR_SETUP_AW, S_WR_SETUP_RETR,
                       S_WR_TX_RF_CH, S_WR_RX_RF_CH, S_WR_RF_SETUP,
                       S_WR_ADDR_P0, S_WR_ADDR_P1, S_WR_ADDR_P2,
                       S_WR_ADDR_P3, S_WR_ADDR_P4, S_WR_ADDR_P5,
                       S_WAIT_CE_ASSERT,
                       S_EXIT);

    signal state    : SM_States := S_IDLE;

    COMPONENT WaitStatesGenerator
    PORT(
        WAITSTATES : IN std_logic_vector(16 downto 0);
        ENABLE : IN std_logic;
        BUSY : OUT std_logic;
        CLK : IN std_logic;
        RESET : IN std_logic
        );
```

```
    END COMPONENT;

    signal  regWaitStates   : std_logic_vector(16 downto 0);
    signal  regWaitEn    : std_logic;
    signal  regWaitBusy     : std_logic;

    signal regCOMMAND   : std_logic_vector(7 downto 0);
    signal regPLDLEN    : std_logic_vector(4 downto 0);
    signal regPLDEN : std_logic;
    signal regPLDWR : std_logic;
    signal regDOUTPLD   : std_logic_vector(7 downto 0);
    signal regDOUTADD   : std_logic_vector(4 downto 0);
    signal regDOUTWEN   : std_logic;
    signal regDINADD    : std_logic_vector(4 downto 0);
    signal regCMDEN : std_logic := '0';
    signal regRXCS      : std_logic := '0';
    signal regTXCS      : std_logic := '0';
    signal regTXCE      : std_logic := '0';
    signal regRXCE      : std_logic := '0';
    signal regPLDSET    : std_logic_vector(11 downto 0);
    signal regDOUT      : std_logic_vector(7 downto 0);
    signal regADD       : std_logic_vector(9 downto 0);
    signal regWEN       : std_logic;
    signal regBUSY      : std_logic;
    signal sWait    : std_logic := '0';

begin
    UWS: WaitStatesGenerator PORT MAP(
        WAITSTATES => regWaitStates,
        ENABLE => regWaitEn,
        BUSY => regWaitBusy,
        CLK => CLK,
        RESET => RESET
    );

    process(CLK, RESET) begin
        if(RESET='1') then
            regCMDEN     <= '0';
            regRXCS      <= '0';
            regRXCE      <= '0';
            regTXCS      <= '0';
            regTXCE      <= '0';
            sWait        <= '0';
            regWaitEn    <= '0';
            state        <= S_IDLE;
        elsif(rising_edge(CLK)) then
            case (state) is
                when S_IDLE =>
                    if(EN='1') then
                        sWait       <= '0';
                        if(OP=x"01") then
                            regCOMMAND  <= x"02";
                            regPLDEN        <= '1';
                            regPLDLEN   <= (others=>'0');
                            regPLDWR        <= '0';
                            regDINADD   <= (others=>'0');
                            regDOUTADD  <= (others=>'0');
                            regDOUTWEN  <= '0';
                            state           <= S_RD_EN_RXADDR;
                        elsif(OP=x"02") then
                            sWait               <= '0';
                            regRXCE         <= '1';
                            regTXCE         <= '1';
                            regWaitStates   <= conv_std_logic_vector(50, 17);
```

```
                state               <= S_WAIT_CE_DEASSERT;
            else
                regWaitStates   <= conv_std_logic_vector(5, 17);
                state               <= S_WAIT_CE_ASSERT;
            end if;
        else
            regCMDEN        <= '0';
            regRXCS     <= '0';
            regRXCE     <= '0';
            regTXCS     <= '0';
            regTXCE     <= '0';
            sWait           <= '0';
            regWaitEn   <= '0';
            regPLDSET   <= (others=>'0');
        end if;

    -- Read configuration sequence
    when S_RD_EN_RXADDR =>
        regWEN  <= '0';
        regADD  <= "1000000000";
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='1') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';
            regDOUT <= DINPLD;
            regWEN  <= '1';
            --Next command setup
            regCOMMAND  <= x"03";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '0';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '0';
            -- Next state
            state       <= S_RD_SETUP_AW;
        end if;
    when    S_RD_SETUP_AW =>
        regWEN  <= '0';
        regADD  <= "1000000001";
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';
            regDOUT <= DINPLD;
            regWEN  <= '1';
            --Next command setup
            regCOMMAND  <= x"04";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '0';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '0';
```

```
            -- Next state
            state       <= S_RD_SETUP_RETR;
        end if;
    when S_RD_SETUP_RETR =>
        regWEN  <= '0';
        regADD  <= "1000000010";
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regTXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regTXCS <= '0';
            sWait       <= '0';
            regDOUT <= DINPLD;
            regWEN  <= '1';
            --Next command setup
            regCOMMAND  <= x"05";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '0';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '0';
            -- Next state
            state       <= S_RD_TX_RF_CH;
        end if;
    when S_RD_TX_RF_CH =>
        regWEN  <= '0';
        regADD  <= "1000000011";
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regTXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regTXCS <= '0';
            sWait       <= '0';
            regDOUT <= DINPLD;
            regWEN  <= '1';
            --Next command setup
            regCOMMAND  <= x"05";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '0';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '0';
            -- Next state
            state       <= S_RD_RX_RF_CH;
        end if;
    when S_RD_RX_RF_CH =>
        regWEN  <= '0';
        regADD  <= "1000000100";
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
```

```
            sWait       <= '0';
            regDOUT <= DINPLD;
            regWEN  <= '1';
            --Next command setup
            regCOMMAND  <= x"06";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '0';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '0';
            -- Next state
            state       <= S_RD_RF_SETUP;
        end if;
    when S_RD_RF_SETUP =>
        regWEN  <= '0';
        regADD  <= "1000000101";
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';
            regDOUT <= DINPLD;
            regWEN  <= '1';
            --Next command setup
            regCOMMAND  <= x"0A";
            regPLDEN        <= '1';
            regPLDLEN   <= conv_std_logic_vector(4,5);
            regPLDWR        <= '0';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '0';
            -- Next state
            state       <= S_RD_ADDR_P0;
        end if;
    when S_RD_ADDR_P0 =>
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regWEN  <= '0';
            regADD  <= "1000000110";
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            regDOUT <= DINPLD;
            regWEN  <= '1';
            if(regADD="1000001010")then
                sWait           <= '0';
                regWEN      <= '0';
                --Next command setup
                regCOMMAND  <= x"0B";
                regPLDEN        <= '1';
                regPLDLEN   <= conv_std_logic_vector(4,5);
                regPLDWR        <= '0';
                regDINADD   <= (others=>'0');
                regDOUTADD  <= (others=>'0');
                regDOUTWEN  <= '0';
                -- Next state
```

```
                state       <= S_RD_ADDR_P1;
            else
                if(regWEN='1') then
                    regADD      <= regADD + '1';
                    regDINADD   <= regDINADD + '1';
                end if;
            end if;
        end if;
    when S_RD_ADDR_P1 =>
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regWEN  <= '0';
            regADD  <= "1000001011";
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            regDOUT <= DINPLD;
            regWEN  <= '1';
            if(regADD="1000001111") then
                sWait           <= '0';
                regWEN      <= '0';
                --Next command setup
                regCOMMAND  <= x"0C";
                regPLDEN        <= '1';
                regPLDLEN   <= (others=>'0');
                regPLDWR        <= '0';
                regDINADD   <= (others=>'0');
                regDOUTADD  <= (others=>'0');
                regDOUTWEN  <= '0';
                -- Next state
                state       <= S_RD_ADDR_P2;
            else
                if(regWEN='1') then
                    regADD      <= regADD + '1';
                    regDINADD   <= regDINADD + '1';
                end if;
            end if;
        end if;
    when S_RD_ADDR_P2 =>
        regWEN  <= '0';
        regADD  <= "1000010000";
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';
            regDOUT <= DINPLD;
            regWEN  <= '1';
            --Next command setup
            regCOMMAND  <= x"0D";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '0';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '0';
            -- Next state
```

```
            state       <= S_RD_ADDR_P3;
        end if;
    when S_RD_ADDR_P3 =>
        regWEN  <= '0';
        regADD  <= "1000010001";
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';
            regDOUT <= DINPLD;
            regWEN  <= '1';
            --Next command setup
            regCOMMAND  <= x"0E";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '0';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '0';
            -- Next state
            state       <= S_RD_ADDR_P4;
        end if;
    when S_RD_ADDR_P4 =>
        regWEN  <= '0';
        regADD  <= "1000010010";
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';
            regDOUT <= DINPLD;
            regWEN  <= '1';
            --Next command setup
            regCOMMAND  <= x"0F";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '0';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '0';
            -- Next state
            state       <= S_RD_ADDR_P5;
        end if;
    when S_RD_ADDR_P5 =>
        regWEN  <= '0';
        regADD  <= "1000010011";
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS <= '1';
            regCMDEN    <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN    <= '0';
            sWait       <= '1';
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';
```

```
            regDOUT <= DINPLD;
            regWEN  <= '1';
            --Next command setup
            regCOMMAND  <= x"FF";
            regPLDEN        <= '0';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '0';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '0';
            -- Next state
            state       <= S_STALL;
        end if;
    when S_STALL =>
        regWEN  <= '0';
        state       <= S_EXIT;
    -- Write configuration sequence
    when S_WAIT_CE_DEASSERT =>
        regPLDSET   <= (others=>'0');
        regWEN      <= '0';
        regADD      <= "1000000000";
        if((regWaitEn='0') and (regWaitBusy='0') and (sWait='0')) then
            regWaitEn   <= '1';
        elsif((regWaitEn='1') and (regWaitBusy='1') and (sWait='0')) then
            regWaitEn   <= '0';
            sWait           <= '1';
        elsif((regWaitEn='0') and (regWaitBusy='0') and (sWait='1')) then
            sWait               <= '0';
            regCOMMAND  <= x"22";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '1';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '0';
            state   <= S_WR_EN_RXADDR;
        end if;
    when S_WR_EN_RXADDR =>
        regDOUTPLD  <= DIN;
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS     <= '1';
            regCMDEN        <= '1';
            regDOUTWEN  <= '1';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN        <= '0';
            regDOUTWEN  <= '0';
            sWait           <= '1';
            regADD      <= "1000000001";
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS     <= '0';
            sWait           <= '0';
            regCOMMAND  <= x"23";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '1';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '1';
            state   <= S_WR_SETUP_AW;
        end if;
    when S_WR_SETUP_AW =>
        regDOUTPLD  <= DIN;
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS     <= '1';
```

```
            regTXCS     <= '1';
            regCMDEN        <= '1';
            regDOUTWEN  <= '0';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN        <= '0';
            regDOUTWEN  <= '0';
            sWait           <= '1';
            regADD      <= "1000000010";
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS     <= '0';
            regTXCS     <= '0';
            sWait           <= '0';
            regCOMMAND  <= x"24";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '1';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '1';
            state   <= S_WR_SETUP_RETR;
        end if;
    when S_WR_SETUP_RETR =>
        regDOUTPLD  <= DIN;
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regTXCS     <= '1';
            regCMDEN        <= '1';
            regDOUTWEN  <= '0';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN        <= '0';
            regDOUTWEN  <= '0';
            sWait           <= '1';
            regADD      <= "1000000011";
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regTXCS <= '0';
            sWait       <= '0';
            regCOMMAND  <= x"25";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '1';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '1';
            state   <= S_WR_TX_RF_CH;
        end if;
    when S_WR_TX_RF_CH =>
        regDOUTPLD  <= DIN;
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regTXCS     <= '1';
            regCMDEN        <= '1';
            regDOUTWEN  <= '0';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN        <= '0';
            regDOUTWEN  <= '0';
            sWait           <= '1';
            regADD      <= "1000000100";
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regTXCS <= '0';
            sWait       <= '0';
            regCOMMAND  <= x"25";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '1';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
```

```
            regDOUTWEN  <= '1';
            state   <= S_WR_RX_RF_CH;
        end if;
    when S_WR_RX_RF_CH =>
        regDOUTPLD  <= DIN;
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS     <= '1';
            regCMDEN        <= '1';
            regDOUTWEN  <= '0';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN        <= '0';
            regDOUTWEN  <= '0';
            sWait           <= '1';
            regADD      <= "1000000101";
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';
            regCOMMAND  <= x"26";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '1';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '1';
            state   <= S_WR_RF_SETUP;
        end if;
    when S_WR_RF_SETUP =>
        regDOUTPLD  <= DIN;
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS     <= '1';
            regTXCS     <= '1';
            regCMDEN        <= '1';
            regDOUTWEN  <= '0';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN        <= '0';
            regDOUTWEN  <= '0';
            sWait           <= '1';
            regADD      <= "1000000110";
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            regTXCS <= '0';
            if(regDOUTWEN='1') then
                regDOUTADD  <= regDOUTADD + '1';
            else
                regDOUTWEN  <= '1';
            end if;
            if(regADD="1000001011") then
                sWait       <= '0';
                regDOUTWEN  <= '0';
                regCOMMAND  <= x"2A";
                regPLDEN        <= '1';
                regPLDLEN   <= conv_std_logic_vector(4,5);
                regPLDWR        <= '1';
                regDINADD   <= (others=>'0');
                regDOUTADD  <= (others=>'0');
                state   <= S_WR_ADDR_P0;
            else
                if(regDOUTWEN='1') then
                    regADD  <= regADD + '1';
                end if;
            end if;
        end if;
    when S_WR_ADDR_P0 =>
        regDOUTPLD  <= DIN;
```

```
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS     <= '1';
            regCMDEN        <= '1';
            regDOUTWEN  <= '0';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN        <= '0';
            regDOUTWEN  <= '0';
            sWait           <= '1';
            regADD      <= "1000001011";
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            if(regDOUTWEN='1') then
                regDOUTADD  <= regDOUTADD + '1';
            else
                regDOUTWEN  <= '1';
            end if;
            if(regADD="1000010000") then
                sWait       <= '0';
                regDOUTWEN  <= '0';
                regCOMMAND  <= x"2B";
                regPLDEN        <= '1';
                regPLDLEN   <= conv_std_logic_vector(4,5);
                regPLDWR        <= '1';
                regDINADD   <= (others=>'0');
                regDOUTADD  <= (others=>'0');
                state   <= S_WR_ADDR_P1;
            else
                if(regDOUTWEN='1') then
                    regADD  <= regADD + '1';
                end if;
            end if;
        end if;
    when S_WR_ADDR_P1 =>
        regDOUTPLD  <= DIN;
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS     <= '1';
            regCMDEN        <= '1';
            regDOUTWEN  <= '0';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN        <= '0';
            regDOUTWEN  <= '0';
            sWait           <= '1';
            regADD      <= "1000010000";
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';
            regCOMMAND  <= x"2C";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '1';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '1';
            state   <= S_WR_ADDR_P2;
        end if;
    when S_WR_ADDR_P2 =>
        regDOUTPLD  <= DIN;
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS     <= '1';
            regCMDEN        <= '1';
            regDOUTWEN  <= '0';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN        <= '0';
            regDOUTWEN  <= '0';
```

```
            sWait           <= '1';
            regADD      <= "1000010001";
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';
            regCOMMAND  <= x"2D";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '1';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '1';
            state   <= S_WR_ADDR_P3;
        end if;
    when S_WR_ADDR_P3 =>
        regDOUTPLD  <= DIN;
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS     <= '1';
            regCMDEN        <= '1';
            regDOUTWEN  <= '0';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN        <= '0';
            regDOUTWEN  <= '0';
            sWait           <= '1';
            regADD      <= "1000010010";
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';
            regCOMMAND  <= x"2E";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '1';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '1';
            state   <= S_WR_ADDR_P4;
        end if;
    when S_WR_ADDR_P4 =>
        regDOUTPLD  <= DIN;
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS     <= '1';
            regCMDEN        <= '1';
            regDOUTWEN  <= '0';
        elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
            regCMDEN        <= '0';
            regDOUTWEN  <= '0';
            sWait           <= '1';
            regADD      <= "1000010011";
        elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
            regRXCS <= '0';
            sWait       <= '0';
            regCOMMAND  <= x"2F";
            regPLDEN        <= '1';
            regPLDLEN   <= (others=>'0');
            regPLDWR        <= '1';
            regDINADD   <= (others=>'0');
            regDOUTADD  <= (others=>'0');
            regDOUTWEN  <= '1';
            state   <= S_WR_ADDR_P5;
        end if;
    when S_WR_ADDR_P5 =>
        regDOUTPLD  <= DIN;
        if((regCMDEN='0') and (CMDBUSY='0') and (sWait='0')) then
            regRXCS     <= '1';
```

```
                    regCMDEN        <= '1';
                    regDOUTWEN  <= '0';
                elsif((regCMDEN='1') and (CMDBUSY='1') and (sWait='0')) then
                    regCMDEN        <= '0';
                    regDOUTWEN  <= '0';
                    sWait           <= '1';
                elsif((regCMDEN='0') and (CMDBUSY='0') and (sWait='1')) then
                    regRXCS         <= '0';
                    sWait               <= '0';
                    regWaitStates   <= conv_std_logic_vector(50, 17);
                    state               <= S_WAIT_CE_ASSERT;
                end if;
            when S_WAIT_CE_ASSERT =>
                if((regWaitEn='0') and (regWaitBusy='0') and (sWait='0')) then
                    regWaitEn   <= '1';
                elsif((regWaitEn='1') and (regWaitBusy='1') and (sWait='0')) then
                    regWaitEn   <= '0';
                    sWait           <= '1';
                elsif((regWaitEn='0') and (regWaitBusy='0') and (sWait='1')) then
                    sWait       <= '0';
                    regTXCE <= '0';
                    regRXCE <= '0';
                    state   <= S_EXIT;
                end if;
            when S_EXIT =>
                regWEN  <= '0';
                if(EN='0') then
                    state   <= S_IDLE;
                end if;
            when others =>
                regCMDEN        <= '0';
                regRXCS     <= '0';
                regRXCE     <= '0';
                regTXCS     <= '0';
                regTXCE     <= '0';
                sWait           <= '0';
                regWaitEn   <= '0';
                state           <= S_IDLE;
        end case;
    end if;
end process;

regBUSY <= '0' when ((state=S_IDLE) or (state=S_EXIT)) else '1';

COMMAND <= regCOMMAND   when (regBUSY='1') else (others=>'Z');
PLDLEN  <= regPLDLEN    when (regBUSY='1') else (others=>'Z');
PLDEN       <= regPLDEN     when (regBUSY='1') else 'Z';
PLDWR       <= regPLDWR     when (regBUSY='1') else 'Z';
DOUTPLD <= regDOUTPLD   when (regBUSY='1') else (others=>'Z');
DOUTADD <= regDOUTADD   when (regBUSY='1') else (others=>'Z');
DOUTWEN <= regDOUTWEN   when (regBUSY='1') else 'Z';
DINADD  <= regDINADD    when (regBUSY='1') else (others=>'Z');
CMDEN       <= regCMDEN     when (regBUSY='1') else 'Z';
RXCS        <= regRXCS      when (regBUSY='1') else 'Z';
TXCS        <= regTXCS      when (regBUSY='1') else 'Z';
TXCE        <= regTXCE      when (regBUSY='1') else 'Z';
RXCE        <= regRXCE      when (regBUSY='1') else 'Z';
PLDSET  <= regPLDSET    when (regBUSY='1') else (others=>'Z');
DOUT        <= regDOUT      when (regBUSY='1') else (others=>'Z');
ADD     <= regADD       when (regBUSY='1') else (others=>'Z');
WEN     <= regWEN       when (regBUSY='1') else 'Z';

BUSY        <= regBUSY;
end Behavioral;
```

4.9 Command handler

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RfCmdParser is
    Port(
        TXDATA  : out STD_LOGIC_VECTOR(7 downto 0);
        RXDATA  : in  STD_LOGIC_VECTOR(7 downto 0);
        SPIWR       : out STD_LOGIC;
        SPIOE       : out STD_LOGIC;
        SPIBUSY : in  STD_LOGIC;
        STATUS  : out STD_LOGIC_VECTOR(7 downto 0);
        COMMAND : in  STD_LOGIC_VECTOR(7 downto 0);
        PLDLEN  : in  STD_LOGIC_VECTOR(4 downto 0);
        PLDEN       : in  STD_LOGIC;
        PLDWR       : in  STD_LOGIC;
        DINPLD  : in  STD_LOGIC_VECTOR(7 downto 0);
        DINADD  : in  STD_LOGIC_VECTOR(4 downto 0);
        DINWEN  : in  STD_LOGIC;
        DOUTPLD : out STD_LOGIC_VECTOR(7 downto 0);
        DOUTADD : in  STD_LOGIC_VECTOR(4 downto 0);
        ENABLE  : in  STD_LOGIC;
        BUSY        : out   STD_LOGIC;
        CLK     : in  STD_LOGIC;
      RESET     : in  STD_LOGIC);
end RfCmdParser;

architecture Behavioral of RfCmdParser is

    COMPONENT DP_DRam_32x8
    PORT (
        a       : IN STD_LOGIC_VECTOR(4 downto 0);
        d       : IN STD_LOGIC_VECTOR(7 downto 0);
        dpra    : IN STD_LOGIC_VECTOR(4 downto 0);
        clk : IN STD_LOGIC;
        we      : IN STD_LOGIC;
        spo : OUT STD_LOGIC_VECTOR(7 downto 0);
        dpo : OUT STD_LOGIC_VECTOR(7 downto 0));
    END COMPONENT;

    signal TxPldDin : std_logic_vector(7 downto 0);
    signal TxPldAdd : std_logic_vector(4 downto 0);
    signal RxPldDout    : std_logic_vector(7 downto 0);
    signal RxPldAdd : std_logic_vector(4 downto 0);
    signal RxPldWen : std_logic;

    type SM_States is (S_IDLE,
                       S_CMD_WR, S_CMD_WAIT,
                       S_PLD_WR, S_PLD_WAIT,
                       S_EXIT);

    signal state    : SM_States := S_IDLE;
    signal stsReg       : std_logic_vector(7 downto 0);
    signal cmdReg       : std_logic_vector(7 downto 0);
    signal dlenReg      : std_logic_vector(4 downto 0);
    signal pldEnReg : std_logic;
    signal pldWrReg : std_logic;
    signal byteCnt  : std_logic_vector(4 downto 0);
    signal spiWrEn  : std_logic;
begin
```

```
RRX: DP_DRam_32x8 PORT MAP (
    a => DINADD,
    d => DINPLD,
    dpra => TxPldAdd,
    clk => CLK,
    we => DINWEN,
    dpo => TxPldDin
);

RTX: DP_DRam_32x8 PORT MAP (
    a => RxPldAdd,
    d => RxPldDout,
    dpra => DOUTADD,
    clk => CLK,
    we => RxPldWen,
    dpo => DOUTPLD
);

SPIOE       <= '1';

process(CLK, RESET) begin
    if(RESET='1') then
        state       <= S_IDLE;
        spiWrEn <= '0';
        byteCnt <= (others=>'0');
    elsif(rising_edge(CLK)) then
        case (state) is
            when S_IDLE =>
                spiWrEn <= '0';
                byteCnt <= (others=>'0');
                if((ENABLE='1') and (SPIBUSY='0')) then
                    cmdReg  <= COMMAND;
                    dlenReg <= PLDLEN;
                    pldEnReg    <= PLDEN;
                    pldWrReg    <= PLDWR;
                    state       <= S_CMD_WR;
                end if;
            when S_CMD_WR =>
                byteCnt <= (others=>'0');
                TXDATA  <= cmdReg;
                if(spiWrEn='0') then
                    spiWrEn <= '1';
                else
                    if(SPIBUSY='1') then
                        spiWrEn <= '0';
                        state       <= S_CMD_WAIT;
                    end if;
                end if;
            when S_CMD_WAIT =>
                if(SPIBUSY='0') then
                    STATUS  <= RXDATA;
                    if (pldEnReg='0') then
                        state   <= S_EXIT;
                    else
                        byteCnt <= (others=>'0');
                        state       <= S_PLD_WR;
                    end if;
                end if;
            when S_PLD_WR =>
                RxPldWen        <= '0';
                if(pldWrReg='1') then
                    TXDATA  <= TxPldDin;
                else
                    TXDATA  <= x"FF";
```

```
                end if;
                if(spiWrEn='0') then
                    spiWrEn <= '1';
                else
                    if(SPIBUSY='1') then
                        spiWrEn     <= '0';
                        state   <= S_PLD_WAIT;
                    end if;
                end if;
            when S_PLD_WAIT =>
                RxPldDout   <= RXDATA;
                if(SPIBUSY='0') then
                    if(RxPldWen='0') then
                        RxPldWen        <= '1';
                    end if;
                    if(byteCnt=dlenReg) then
                        if(RxPldWen='1') then
                            RxPldWen    <= '0';
                            state       <= S_EXIT;
                        end if;
                    else
                        if(RxPldWen='1') then
                            RxPldWen    <= '0';
                            byteCnt <= byteCnt + '1';
                            state       <= S_PLD_WR;
                        end if;
                    end if;
                end if;
            when S_EXIT =>
                if(ENABLE='0') then
                    state   <= S_IDLE;
                end if;
            when others =>
                spiWrEn <= '0';
                byteCnt <= (others=>'0');
                if(SPIBUSY='0') then
                    state   <= S_IDLE;
                end if;
        end case;
    end if;
end process;

TxPldAdd    <= byteCnt;
RxPldAdd    <= byteCnt;
SPIWR       <= spiWrEn;
BUSY        <= '0' when ((state=S_IDLE) or (state=S_EXIT)) else '1';
end Behavioral;
```

4.10 SPI master

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RfSpiMaster is
     Generic(
        CLOCKDIV2 : INTEGER := 2;
        DATAWIDTH : integer := 8);
    Port(
      SCK       : out STD_LOGIC;
        MISO        : in  STD_LOGIC;
        MOSI        : out STD_LOGIC;
        TXDATA  : in  STD_LOGIC_VECTOR(DATAWIDTH-1 downto 0);
        WR      : in  STD_LOGIC;
        RXDATA  : out STD_LOGIC_VECTOR(DATAWIDTH-1 downto 0);
        OE      : in  STD_LOGIC;
        BUSY        : out STD_LOGIC;
        CLK     : in  STD_LOGIC;
        RESET       : in  STD_LOGIC);
end RfSpiMaster;

architecture Behavioral of RfSpiMaster is
    type FsmSpiType is (IDLE, CSN_ASSERT, DATA_TRANSFER, CSN_DEASSERT);

    signal FsmState : FsmSpiType := IDLE;
    signal txRegister   : STD_LOGIC_VECTOR(DATAWIDTH-1 downto 0) := (others=>'0');
    signal rxRegister   : STD_LOGIC_VECTOR(DATAWIDTH-1 downto 0) := (others=>'0');
    signal bitsToEnd    : STD_LOGIC_VECTOR(DATAWIDTH-1 downto 0) := (others=>'1');
    signal EnableI      : STD_LOGIC;
    signal EnableQ      : STD_LOGIC;
    signal busyFlag : STD_LOGIC;
    signal misoSignal   : STD_LOGIC;
    signal mosiSignal   : STD_LOGIC;
    signal sckSignal    : STD_LOGIC := '0';

    COMPONENT CLKDIV
    GENERIC(
        CLOCKDIV2: NATURAL := 10);
    PORT(
        CLKIN       : IN  STD_LOGIC;
        CEOUT       : OUT STD_LOGIC;
        CEOUT180    : OUT STD_LOGIC;
        RSTIN       : IN  STD_LOGIC
        );
    END COMPONENT;
begin
    U1: CLKDIV
    GENERIC MAP(
        CLOCKDIV2=> CLOCKDIV2)
    PORT MAP(
        CLKIN       => CLK,
        CEOUT       => EnableI,
        CEOUT180    => EnableQ,
        RSTIN       => RESET
    );

    misoSignal  <= MISO;
    MOSI            <= mosiSignal;
    BUSY            <= '0' when (FsmState=IDLE) else '1';
    RXDATA      <= rxRegister when (OE='1') else (others=>'Z');
    mosiSignal  <= txRegister(DATAWIDTH-1);
```

```
    process(CLK, RESET) begin
        if(RESET='1')then
            FsmState <= IDLE;
        elsif(rising_edge(CLK))then
            if((WR='1') and (FsmState=IDLE))then
                bitsToEnd   <= (others=>'1');
                txRegister  <= TXDATA;
                FsmState        <= CSN_ASSERT;
            else
                case FsmState is
                    when IDLE =>
                        SCK <= '0';
                    when CSN_ASSERT =>
                        SCK <= '0';
                        if(EnableQ = '1')THEN
                            FsmState    <= DATA_TRANSFER;
                        end if;
                    when DATA_TRANSFER =>
                        if(bitsToEnd(DATAWIDTH-1)='1')then
                            if(EnableI='1')then
                                SCK         <= '1';
                                rxRegister  <= rxRegister(DATAWIDTH-2 downto 0) &
misoSignal;
                            elsif(EnableQ='1') then
                                SCK         <= '0';
                                txRegister  <= txRegister(DATAWIDTH-2 downto 0) & '0';
                                bitsToEnd   <= bitsToEnd(DATAWIDTH-2 downto 0) & '0';
                            end if;
                        else
                            FsmState    <= CSN_DEASSERT;
                            SCK     <= '0';
                        end if;
                    when CSN_DEASSERT =>
                        FsmState    <= IDLE;
                        SCK     <= '0';
                    when others =>
                        FsmState    <= IDLE;
                        SCK     <= '0';
                        bitsToEnd<= (others=>'0');
                end case;
            end if;
        end if;
    end process;
end Behavioral;
h
```

5 USB Client Software

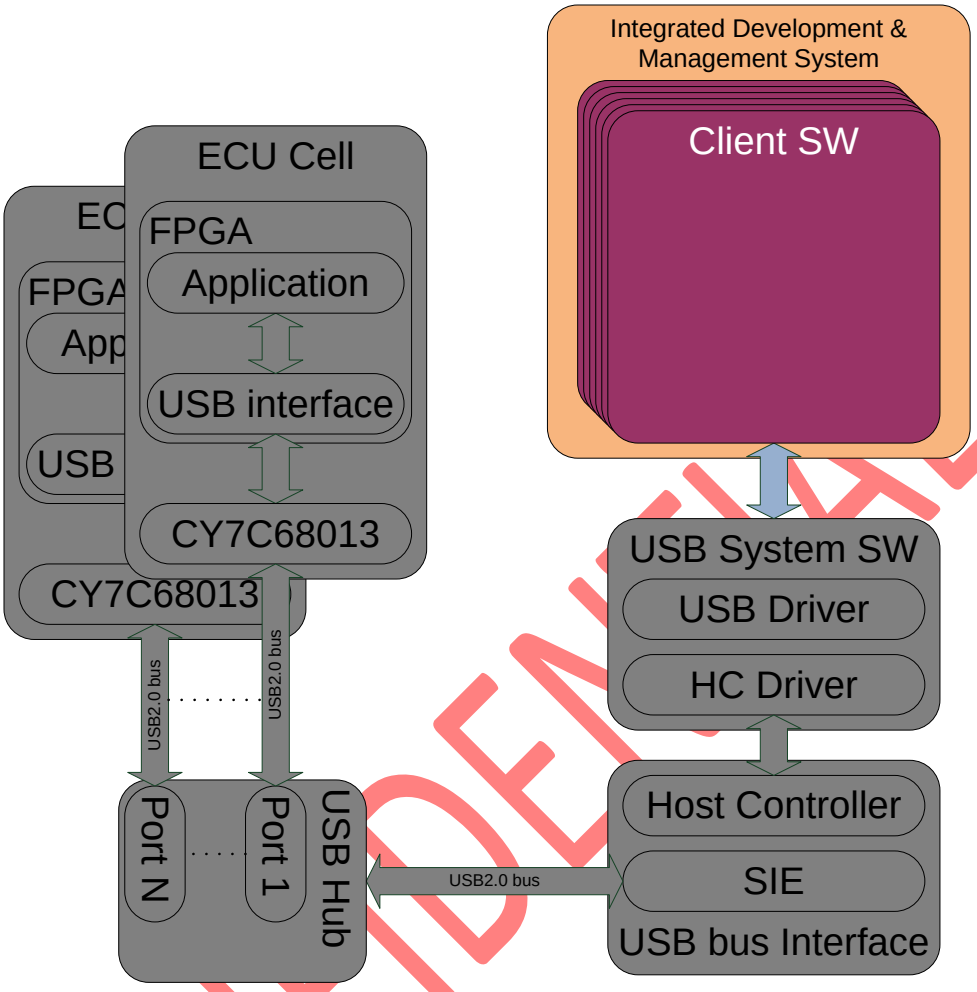


Figure 5 – USB Overview

5.1 Client Software - CyTools Namespace

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Text;
using CyUSB;
using System.Threading;

namespace CyTools
{
    public enum DeviceType
    {
        CyUsb = CyConst.DEVICES_CYUSB,
        Hid = CyConst.DEVICES_HID,
        Msc = CyConst.DEVICES_MSC,
        All = CyConst.DEVICES_CYUSB | CyConst.DEVICES_HID | CyConst.DEVICES_MSC
    }

    public delegate void BytesTransferredEvent(object sender, int Device, int Channel, int
NumBytes);
    public delegate void EpTerminated(object sender, bool In, int Device, int Channel);
    public delegate void DeviceTreeChanged(object sender, CyDeviceInterface Device);
```

5.2 CyWrapper

```
    /// <summary>
    /// Defines the global devices access methods, user handled events and manages device
PnP
    /// </summary>
    public class CyWrapper
    {
        public event BytesTransferredEvent BytesReceived;
        public event BytesTransferredEvent BytesSent;
        public event DeviceTreeChanged DeviceAdded;
        public event DeviceTreeChanged DeviceRemoved;
        public event EpTerminated WorkTerminated;

        private byte devMask;

        USBDeviceList usbDevices;
        App_PnP_Callback evHandler;

        CyDeviceInterface[] ConnectedDevices, DisconnectedDevices;
        int NumConnectedDevices = 0;

        public int Count
        {
            get
            {
                return NumConnectedDevices;
            }
        }

        /// <summary>
        /// Returns device number in array, otherwise returns -1
        /// </summary>
        /// <param name="CyDevice">Device to search</param>
        /// <returns>Returns device number in array, otherwise returns -1</returns>
        public int IndexOf(CyDeviceInterface CyDevice)
        {
            return Array.IndexOf(ConnectedDevices, CyDevice);
        }
```

```
        /// <summary>
        /// The array of devices currently connected
        /// </summary>
        public CyDeviceInterface[] CyDevices
        {
            get
            {
                return ConnectedDevices;
            }
        }

        /// <summary>
        /// Stops all threads and releases all resources
        /// </summary>
        public void Dispose()
        {
            if (ConnectedDevices == null)
            {
                return;
            }
            foreach (CyDeviceInterface var in ConnectedDevices)
            {
                if (var != null)
                {
                    var.Dispose();
                }
            }
            evHandler = null;
            Thread.Sleep(100);
            usbDevices.Dispose();
        }

        /// <summary>
        /// Clears the selected queue
        /// </summary>
        /// <param name="bRx">True if is a reception queue, otherwise false</param>
        /// <param name="Channel">The channel number</param>
        /// <param name="Device">The device number</param>
        /// <returns>Returns true if operation completes succefully, otherwisee
false</returns>
        public bool ClearQueue(bool bRx, int Channel, int Device)
        {

            if (ConnectedDevices != null)
            {
                if (Device < NumConnectedDevices)
                {
                    return ConnectedDevices[Device].ClearQueue(bRx, Channel);
                }
                return false;
            }
            return false;
        }

        /// <summary>
        /// Commits data to send to a specified device
        /// </summary>
        /// <param name="Data">Data to send</param>
        /// <param name="Length">Length of data to send</param>
        /// <param name="Channel">The channel number</param>
        /// <param name="Device">The device number</param>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
```

```
        public bool SendBytes(byte[] Data, int Length, int Channel, int Device)
        {
            if ((ConnectedDevices != null)&&(Device < NumConnectedDevices))
            {
                if (Device < NumConnectedDevices)
                {
                    return ConnectedDevices[Device].SendBytes(Data, Length, Channel);
                }
                return false;
            }
            return false;
        }

        /// <summary>
        /// Commits data to send to a specified device
        /// </summary>
        /// <param name="Data">Data to send</param>
        /// <param name="Length">Length of data to send</param>
        /// <param name="Channel">The channel number</param>
        /// <param name="SerialNumber">The device serial number</param>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool SendBytes(byte[] Data, int Length, int Channel, string SerialNumber)
        {
            foreach (CyDeviceInterface var in ConnectedDevices)
            {
                if (var.ToString() == SerialNumber)
                {
                    return var.SendBytes(Data, Length, Channel);
                }
            }
            return false;
        }

        /// <summary>
        /// Retrieve data received from a specified device
        /// </summary>
        /// <param name="Data">Data received</param>
        /// <param name="Length">Length of data to receive</param>
        /// <param name="Channel">The channel number</param>
        /// <param name="Device">The device number</param>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool GetBytes(out byte[] Data, ref int Length, int Channel, int Device)
        {
            if (ConnectedDevices != null)
            {
                if (Device < NumConnectedDevices)
                {
                    return ConnectedDevices[Device].GetBytes(out Data, ref Length, Channel);
                }
                Data = null;
                return false;
            }
            Data = null;
            return false;
        }

        private void PnP_Event_Handler(IntPtr pnpEvent, IntPtr hRemovedDevice)
        {
            if (evHandler != null)
            {
                if (pnpEvent.Equals(CyConst.DBT_DEVICEREMOVECOMPLETE))
                {
```

```
                USBDeviceList aux = new USBDeviceList(devMask, null);
                bool exists = false;
                CyFX2Device Removed = null;
                foreach (CyUSBDevice var1 in usbDevices)
                {
                    foreach (CyUSBDevice var2 in aux)
                    {
                        if (var1 == var2)
                        {
                            exists = true;
                        }
                    }
                    if (!exists)
                    {
                        if (var1 is CyFX2Device)
                        {
                            Removed = (CyFX2Device)var1;
                        }
                    }
                }
                usbDevices.Remove(hRemovedDevice);
                // Other removal event handling
                UpdateDevices();
                if (Removed != null)
                {
                    StopDevice(Removed);
                }
            }

            if (pnpEvent.Equals(CyConst.DBT_DEVICEARRIVAL))
            {
                usbDevices.Add();
                // Other arrival event handling
                UpdateDevices();
            }
        }
    }

    /// <summary>
    /// Creates a new object using the default filter - CYUSB devices
    /// </summary>
    public CyWrapper()
    {
    bool bStarted = false;

    /// <summary>
    /// Indicates if threads are running or not
    /// </summary>
    public bool Started
    {
        get
        {
            return bStarted;
        }
    }

    /// <summary>
    /// This start automatic listening and management
    /// </summary>
    ///
    public void Start()
    {
        if (bStarted)
            return;
```

```
            devMask = CyConst.DEVICES_CYUSB;
            evHandler = new App_PnP_Callback(PnP_Event_Handler);
            usbDevices = new USBDeviceList(devMask, evHandler);
            UpdateDevices();
            bStarted = true;
        }

        public void Stop()
        {
            if (!bStarted)
                return;
            devMask = 0;
            evHandler = null;
            usbDevices = new USBDeviceList(devMask, evHandler);
            UpdateDevices();
            bStarted = false;
        }

        /// <summary>
        /// Creates a new object using the specified filter
        /// </summary>
        /// <param name="Type">the specified filter</param>
        public CyWrapper(DeviceType Type)
        {
        }

        private bool StopDevice(CyFX2Device Device)
        {
            bool retval = false;
            if (DisconnectedDevices != null)
            {
                foreach (CyDeviceInterface var in DisconnectedDevices)
                {
                    if (var.Device == Device)
                    {
                        retval |= var.Stop();
                        var.Dispose();
                    }
                }
                DisconnectedDevices = null;
            }
            return retval;
        }

        /// <summary>
        /// Suspends all Rx/Tx operations
        /// </summary>
        /// <param name="Device">The device to suspend</param>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool SuspendDevice(CyDeviceInterface Device)
        {
            return SuspendDevice(Device.Device);
        }

        /// <summary>
        /// Suspends all Rx/Tx operations
        /// </summary>
        /// <param name="Device">The device to suspend</param>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        private bool SuspendDevice(CyFX2Device Device)
        {
            bool retval = false;
```

```
            foreach (CyDeviceInterface var in ConnectedDevices)
            {
                if (var.Device == Device)
                {
                    retval |= var.Suspend();
                }
            }
            return retval;
        }

        /// <summary>
        /// Resumes all Rx/Tx operations
        /// </summary>
        /// <param name="Device">The device to resume</param>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool ResumeDevice(CyDeviceInterface Device)
        {
            return ResumeDevice(Device.Device);
        }

        /// <summary>
        /// Resumes all Rx/Tx operations
        /// </summary>
        /// <param name="Device">The device to resume</param>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        private bool ResumeDevice(CyFX2Device Device)
        {
            bool retval = false;
            foreach (CyDeviceInterface var in ConnectedDevices)
            {
                if (var.Device == Device)
                {
                    retval |= var.Resume();
                }
            }
            return retval;
        }

        /// <summary>
        /// Forces the update of the connected devices array
        /// </summary>
        public void UpdateDevices()
        {
            int nullEntry = 0;
            if (ConnectedDevices != null)
            {
                for (int i = 0; i < ConnectedDevices.Length; i++)
                {
                    bool Exists = false;
                    foreach (USBDevice deviceInUsb in usbDevices)
                    {
                        if (deviceInUsb is CyFX2Device)
                        {
                            if (ConnectedDevices[i] != null)
                            {
                                if (ConnectedDevices[i].Device ==
((CyFX2Device)deviceInUsb))
                                {
                                    Exists = true;
                                }
                            }
                            else
```

```
                        {
                            Exists = true;
                        }
                    }
                }
                if (!Exists)
                {
                    if ((DeviceRemoved != null) && (ConnectedDevices[i] != null))
                    {
                        DeviceRemoved(this, ConnectedDevices[i]);
                    }
                    if (DisconnectedDevices == null)
                    {
                        DisconnectedDevices = new CyDeviceInterface[0];
                    }
                    Array.Resize<CyDeviceInterface>(ref DisconnectedDevices,
DisconnectedDevices.Length + 1);
                    DisconnectedDevices[DisconnectedDevices.Length - 1] =
ConnectedDevices[i];
                }
            }

            if (DisconnectedDevices != null)
            {
                foreach (CyDeviceInterface var in DisconnectedDevices)
                {
                    int varIndex = Array.IndexOf(ConnectedDevices, var);
                    if (varIndex >= 0)
                    {
                        for (int i = varIndex; i < ConnectedDevices.Length - 1 -
nullEntry; i++)
                        {
                            ConnectedDevices[i] = ConnectedDevices[i + 1];
                        }
                        nullEntry++;
                    }
                }
                Array.Resize<CyDeviceInterface>(ref ConnectedDevices,
ConnectedDevices.Length - nullEntry);

            }
            nullEntry = 0;
        }
        if (DisconnectedDevices != null)
        {
            foreach (CyDeviceInterface var in DisconnectedDevices)
            {
                var.Stop();
                var.Dispose();
            }
        }
        DisconnectedDevices = null;

        foreach (USBDevice deviceInUsb in usbDevices)
        {
            bool isToAdd = true;
            if (deviceInUsb is CyFX2Device)
            {
                if (ConnectedDevices != null)
                {
                    foreach (CyDeviceInterface deviceInList in ConnectedDevices)
                    {
                        if (deviceInList != null)
                        {
```

```
                                if (deviceInList.Device == deviceInUsb)
                                {
                                    isToAdd = false;
                                }
                            }
                        }
                    }
                    else
                    {
                        isToAdd = false;
                    }
                    if (isToAdd)
                    {
                        if (nullEntry > 0)
                        {
                            ConnectedDevices[ConnectedDevices.Length - nullEntry] = new
CyDeviceInterface((CyFX2Device)deviceInUsb);
                            ConnectedDevices[ConnectedDevices.Length - nullEntry].BytesReceived
+= new BytesTransferredEvent(CyWrapper_BytesReceived);
                            ConnectedDevices[ConnectedDevices.Length - nullEntry].BytesSent +=
new BytesTransferredEvent(CyWrapper_BytesSent);
                            ConnectedDevices[ConnectedDevices.Length -
nullEntry].WorkTerminated += new EpTerminated(CyWrapper_WorkTerminated);
                            if (DeviceAdded != null)
                            {
                                DeviceAdded(this, ConnectedDevices[ConnectedDevices.Length -
nullEntry]);
                            }
                            nullEntry--;
                        }
                        else
                        {
                            if (ConnectedDevices != null)
                            {
                                Array.Resize<CyDeviceInterface>(ref ConnectedDevices,
ConnectedDevices.Length + 1);
                            }
                            else
                            {
                                ConnectedDevices = new CyDeviceInterface[1];
                            }
                            ConnectedDevices[ConnectedDevices.Length - 1] = new
CyDeviceInterface((CyFX2Device)deviceInUsb);
                            ConnectedDevices[ConnectedDevices.Length - 1].BytesReceived += new
BytesTransferredEvent(CyWrapper_BytesReceived);
                            ConnectedDevices[ConnectedDevices.Length - 1].BytesSent += new
BytesTransferredEvent(CyWrapper_BytesSent);
                            ConnectedDevices[ConnectedDevices.Length - 1].WorkTerminated += new
EpTerminated(CyWrapper_WorkTerminated);
                            if (DeviceAdded != null)
                            {
                                DeviceAdded(this, ConnectedDevices[ConnectedDevices.Length -
1]);
                            }
                        }
                    }
                }
                if (nullEntry > 0)
                {
                    Array.Resize<CyDeviceInterface>(ref ConnectedDevices,
ConnectedDevices.Length - nullEntry);
                    nullEntry = 0;
                }
```

```
            if (ConnectedDevices != null)
                NumConnectedDevices = ConnectedDevices.Length;
            else
                NumConnectedDevices = 0;
        }

        void CyWrapper_WorkTerminated(object sender, bool In, int Device, int Channel)
        {
            if (DisconnectedDevices != null)
            {
                if (DisconnectedDevices.Length > 0)
                {
                    int aux = Array.IndexOf(DisconnectedDevices, sender);
                    if (aux < 0)
                    {
                        return;
                    }
                    if ((DisconnectedDevices[aux].EpInCount +
DisconnectedDevices[aux].EpOutCount) == 0)
                    {
                        if ((aux >= 0) && (Channel >= 0))
                        {
                            if (WorkTerminated != null)
                            {
                                WorkTerminated(this, In, aux, Channel);
                            }
                        }
                        DisconnectedDevices[aux].Dispose();
                        for (int i = aux; i < DisconnectedDevices.Length - 1; i++)
                        {
                            DisconnectedDevices[i] = DisconnectedDevices[i + 1];
                        }
                        Array.Resize<CyDeviceInterface>(ref DisconnectedDevices,
DisconnectedDevices.Length - 1);
                    }
                }
            }
        }

        void CyWrapper_BytesSent(object sender, int Device, int Channel, int NumBytes)
        {
            if (ConnectedDevices != null)
            {
                if (ConnectedDevices.Length > 0)
                {
                    int aux = Array.IndexOf(ConnectedDevices, sender);
                    if ((aux >= 0) && (Channel >= 0))
                    {
                        if (BytesSent != null)
                        {
                            BytesSent(this, aux, Channel, NumBytes);
                        }
                    }
                }
            }
        }


        private SortedList<int, long> lRxBytesCounter = new SortedList<int, long>();
        void CyWrapper_BytesReceived(object sender, int Device, int Channel, int NumBytes)
        {
            if (ConnectedDevices != null)
            {
```

```
            if (lRxBytesCounter.Count > ConnectedDevices.Length)
            {
                lRxBytesCounter.Clear();
            }
            if (ConnectedDevices.Length > 0)
            {
                int aux = Array.IndexOf(ConnectedDevices, sender);
                if ((aux >= 0) && (Channel >= 0))
                {
                    if (lRxBytesCounter.ContainsKey(aux))
                        lRxBytesCounter[aux] += NumBytes;
                    else
                        lRxBytesCounter.Add(aux, NumBytes);

                    if (BytesReceived != null)
                    {
                        BytesReceived(this, aux, Channel, NumBytes);
                    }
                }
            }
        }
    }
}
```

5.3 CyDeviceInterface

```
public delegate void StatsEvent(object sender, long TxBw, long RxBw);
/// <summary>
/// Defines a Cypress device access methods, managed resources and events
/// </summary>
public class CyDeviceInterface
{
    public event BytesTransferredEvent BytesReceived;
    public event BytesTransferredEvent BytesSent;
    public event EpTerminated WorkTerminated;
    public event StatsEvent StatsChanged;

    private const int PKTSZ = 512;

    private const int DEPQUEUESZ = 8;
    private const int DEPBUFSZ = PKTSZ * 32;
    private const int DEPQUEUETO = 500;

    private const int CEPQUEUESZ = 8;
    private const int CEPBUFSZ = PKTSZ;
    private const int CEPQUEUETO = 500;

    Queue[] TxQueue, RxQueue, SyncTxQueue, SyncRxQueue;
    CyFX2Device AssociatedDevice;
    int NumInEp = 0, NumOutEp = 0;
    CyInEndpointInterface[] EpIn;
    CyOutEndpointInterface[] EpOut;
    string SerialNumber;

    private Mutex mutTxQueueAccess, mutRxQueueAccess;

    private bool Suspended = false;

    /// <summary>
    /// Returns an list with the queue widths
    /// </summary>
    public List<uint> InEpQueueWidths
    {
```

```
        get
        {
            List<uint> bufsz = new List<uint>();
            for (int i = 0; i < NumInEp; i++)
            {
                bufsz.Add(EpIn[i].QueueWd);
            }
            return bufsz;
        }
    }

    /// <summary>
    /// Returns an list with the queue widths
    /// </summary>
    public List<uint> OutEpQueueWidths
    {
        get
        {
            List<uint> bufsz = new List<uint>();
            for (int i = 0; i < NumOutEp; i++)
            {
                bufsz.Add(EpOut[i].QueueWd);
            }
            return bufsz;
        }
    }

    /// <summary>
    /// Returns an list with the queue depths
    /// </summary>
    public List<uint> InEpQueueDepths
    {
        get
        {
            List<uint> bufsz = new List<uint>();
            for (int i = 0; i < NumInEp; i++)
            {
                bufsz.Add(EpIn[i].QueueSz);
            }
            return bufsz;
        }
    }

    /// <summary>
    /// Returns an list with the queue depths
    /// </summary>
    public List<uint> OutEpQueueDepths
    {
        get
        {
            List<uint> bufsz = new List<uint>();
            for (int i = 0; i < NumOutEp; i++)
            {
                bufsz.Add(EpOut[i].QueueSz);
            }
            return bufsz;
        }
    }

    /// <summary>
    /// Returns an list with the queue timeouts
    /// </summary>
    public List<uint> InEpQueueTimeouts
    {
```

```
        get
        {
            List<uint> bufsz = new List<uint>();
            for (int i = 0; i < NumInEp; i++)
            {
                bufsz.Add(EpIn[i].QueueTo);
            }
            return bufsz;
        }
    }

    /// <summary>
    /// Returns an list with the queue timoeuts
    /// </summary>
    public List<uint> OutEpQueueTimeouts
    {
        get
        {
            List<uint> bufsz = new List<uint>();
            for (int i = 0; i < NumOutEp; i++)
            {
                bufsz.Add(EpOut[i].QueueTo);
            }
            return bufsz;
        }
    }

    /// <summary>
    /// Returns a string representing the current device
    /// </summary>
    /// <returns>Returns a string representing the current device</returns>
    public override string ToString()
    {
        return SerialNumber;
    }

    /// <summary>
    /// Returns true if the device is suspended, otherwise returns false
    /// </summary>
    public bool IsSuspended
    {
        get
        {
            return Suspended;
        }
    }

    /// <summary>
    /// Returns the number of In enpoints (Rx channels)
    /// </summary>
    public int EpInCount
    {
        get
        {
            return NumInEp;
        }
    }

    /// <summary>
    /// Returns the number of Out enpoints (Tx channels)
    /// </summary>
    public int EpOutCount
    {
        get
```

```
            {
                return NumOutEp;
            }
        }

        /// <summary>
        /// Returns the device associated with the current wrapper
        /// </summary>
        public CyFX2Device Device
        {
            get
            {
                return AssociatedDevice;
            }
        }

        /// <summary>
        /// Clears associated queue
        /// </summary>
        /// <param name="bRx">If true clears Rx queue, otherwise clears Tx queue</param>
        /// <param name="Channel">The channel of the current device to clear</param>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool ClearQueue(bool bRx, int Channel)
        {
            if (bRx)
            {
                if (Channel >= NumInEp)
                {
                    return false;
                }
                lock (RxQueue.SyncRoot)
                {
                    lock (RxQueue[Channel].SyncRoot)
                    {
                        RxQueue[Channel].Clear();
                    }
                }
                return true;
            }
            else
            {
                if (Channel >= NumOutEp)
                {
                    return false;
                }
                lock (TxQueue.SyncRoot)
                {
                    lock (TxQueue[Channel].SyncRoot)
                    {
                        TxQueue[Channel].Clear();
                    }
                }
                return true;
            }
        }

        /// <summary>
        /// Sends data to the selected channel
        /// </summary>
        /// <param name="Data">Data to send</param>
        /// <param name="Length">Data length</param>
        /// <param name="Channel">Channel number</param>
```

```
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool SendBytes(byte[] Data, int Length, int Channel)
        {
            if (Channel >= NumOutEp)
            {
                return false;
            }
            if (Length == 0)
            {
                return false;
            }
            if (Data == null)
            {
                return false;
            }
            if (Data.Length < Length)
            {
                return false;
            }

            try
            {
                byte[] aux = new byte[EpOut[Channel].QueueWd];
                int BytesSent = 0;
                while (BytesSent < Data.Length)
                {
                    if ((Data.Length - BytesSent) >= EpOut[Channel].QueueWd)
                    {
                        Array.Copy(Data, BytesSent, aux, 0, EpOut[Channel].QueueWd);
                        lock (SyncTxQueue.SyncRoot)
                        {
                            lock (SyncTxQueue[Channel].SyncRoot)
                            {
                                TxQueue[Channel].Enqueue((byte[])aux.Clone());
                            }
                        }
                        BytesSent += (int)EpOut[Channel].QueueWd;
                    }
                    else
                    {
                        Array.Copy(Data, BytesSent, aux, 0, Data.Length - BytesSent);
                        Array.Resize<byte>(ref aux, Data.Length - BytesSent);
                        lock (SyncTxQueue.SyncRoot)
                        {
                            lock (SyncTxQueue[Channel].SyncRoot)
                            {
                                TxQueue[Channel].Enqueue((byte[])aux.Clone());
                            }
                        }
                        BytesSent += (Data.Length - BytesSent);
                    }

                }
            }
            catch (Exception)
            {
                return false;
            }
            return true;
        }

        byte[] RemainingBytes;
        int RemainingBytesLength;
```

```
        /// <summary>
        /// Gets data from the selected channel
        /// </summary>
        /// <param name="Data">Data received</param>
        /// <param name="Length">Data length</param>
        /// <param name="Channel">Channel number</param>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool GetBytes(out byte[] Data, ref int Length, int Channel)
        {
            byte[] AuxData;
            int AuxDataLength;

            if (Channel >= NumInEp)
            {
                Data = null;
                Length = 0;
                return false;
            }
            if (RxQueue[Channel].Count == 0 && RemainingBytesLength == 0)
            {
                Data = null;
                Length = 0;
                return false;
            }
            if (Length == 0)
            {
                Data = null;
                return false;
            }
            AuxData = new byte[Length];
            AuxDataLength = 0;

            if (RemainingBytesLength > 0)
            {
                if (Length <= RemainingBytesLength)
                {
                    Array.Copy(RemainingBytes, RemainingBytes.Length - RemainingBytesLength,
AuxData, 0, Length);
                    RemainingBytesLength -= Length;
                    Data = AuxData;
                    return true;
                }
                else
                {
                    Array.Copy(RemainingBytes, RemainingBytes.Length - RemainingBytesLength,
AuxData, 0, RemainingBytesLength);
                    AuxDataLength = RemainingBytesLength;
                    RemainingBytesLength = 0;
                }
            }

            if (RxQueue[Channel].Count > 0)
            {
                byte[] aux;
                do
                {
                    lock (SyncRxQueue.SyncRoot)
                    {
                        lock (SyncRxQueue[Channel].SyncRoot)
                        {
                            aux = (byte[])RxQueue[Channel].Dequeue();
                        }
                    }
```

```
                    if (AuxDataLength + aux.Length <= Length)
                    {
                        Array.Copy(aux, 0, AuxData, AuxDataLength, aux.Length);
                        AuxDataLength += aux.Length;
                    }
                    else
                    {
                        Array.Copy(aux, 0, AuxData, AuxDataLength, Length - AuxDataLength);

                        RemainingBytes = aux;
                        RemainingBytesLength = aux.Length - (Length - AuxDataLength);
                        AuxDataLength = Length;
                    }
                }
                while ((Length > AuxDataLength) && (RxQueue[Channel].Count != 0));
            }

            Length = AuxDataLength;
            if (AuxDataLength == 0)
            {
                Data = null;
                return false;
            }
            Data = AuxData;
            return true;
        }

        /// <summary>
        /// Creates a new object to interface with the specified Device
        /// </summary>
        /// <param name="Device">The device that interfaces with the object</param>
        public CyDeviceInterface(CyFX2Device Device)
        {
            tmrRefreshStats = new System.Timers.Timer(1000);
            tmrRefreshStats.Elapsed += new
System.Timers.ElapsedEventHandler(tmrRefreshStats_Elapsed);
            tmrRefreshStats.Enabled = true;

            if (Device == null)
            {
                throw new Exception("Cannot create interface from a null device!!!");
            }
            AssociatedDevice = Device;
            SerialNumber = Device.SerialNumber;
            mutRxQueueAccess = new Mutex();
            mutTxQueueAccess = new Mutex();

            foreach (CyUSBEndPoint var in AssociatedDevice.EndPoints)
            {
                if (var is CyBulkEndPoint)
                {
                    if (var.bIn)
                    {
                        NumInEp++;
                    }
                    else
                    {
                        NumOutEp++;
                    }
                }
            }

            EpIn = new CyInEndpointInterface[NumInEp];
            EpOut = new CyOutEndpointInterface[NumOutEp];
```

```
            RxQueue = new Queue[NumInEp];
            TxQueue = new Queue[NumOutEp];
            SyncRxQueue = new Queue[NumInEp];
            SyncTxQueue = new Queue[NumOutEp];

            try
            {
                for (int i = 0; i < NumInEp; i++)
                {
                    RxQueue[i] = new Queue(2048, 4f);
                    SyncRxQueue[i] = Queue.Synchronized(RxQueue[i]);
                    if (i == 0)
                    {
                        EpIn[i] = new CyInEndpointInterface(AssociatedDevice,
SyncRxQueue[i], i, CEPQUEUESZ, CEPQUEUETO, CEPBUFSZ);
                    }
                    else
                    {
                        EpIn[i] = new CyInEndpointInterface(AssociatedDevice,
SyncRxQueue[i], i, DEPQUEUESZ, DEPQUEUETO, DEPBUFSZ);
                    }
                    EpIn[i].BytesReceived += new
BytesTransferredEvent(CyDeviceInterface_BytesReceived);
                    EpIn[i].WorkFinished += new
EpTerminated(CyDeviceInterface_WorkFinished);
                }
            }
            catch (Exception ex)
            {
                throw new Exception("Cannot create In Endpoint interface", ex);
            }

            try
            {
                for (int i = 0; i < NumOutEp; i++)
                {
                    TxQueue[i] = new Queue(2048, 4f);
                    SyncTxQueue[i] = Queue.Synchronized(TxQueue[i]);
                    if (i == 0)
                    {
                        EpOut[i] = new CyOutEndpointInterface(AssociatedDevice,
SyncTxQueue[i], i, CEPQUEUESZ, CEPQUEUETO, CEPBUFSZ);
                    }
                    else
                    {
                        EpOut[i] = new CyOutEndpointInterface(AssociatedDevice,
SyncTxQueue[i], i, DEPQUEUESZ, DEPQUEUETO, DEPBUFSZ);
                    }
                    EpOut[i].BytesSent += new
BytesTransferredEvent(CyDeviceInterface_BytesSent);
                    EpOut[i].WorkFinished += new
EpTerminated(CyDeviceInterface_WorkFinished);
                }
            }
            catch (Exception ex)
            {
                throw new Exception("Cannot create Out Endpoint interface", ex);
            }
        }

        void CyDeviceInterface_WorkFinished(object sender, bool In, int Device, int Channel)
        {
            if ((NumInEp + NumOutEp) > 0)
```

```
            {
                int aux = -1;
                if ((In) && (NumInEp > 0))
                {
                    aux = Array.IndexOf(EpIn, sender);
                    NumInEp--;
                }
                if ((!In) && (NumOutEp > 0))
                {
                    aux = Array.IndexOf(EpOut, sender);
                    NumOutEp--;
                }
                if ((aux >= 0) && ((NumOutEp + NumInEp) == 0))
                {
                    WorkTerminated(this, In, -1, aux);
                }
            }
        }

        private long fTxBw, fRxBw;

        System.Timers.Timer tmrRefreshStats;
        DateTime tStart = DateTime.Now;
        void tmrRefreshStats_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
        {
            TimeSpan tSpan = DateTime.Now - tStart;
            fTxBw = (long)(lTxBytesCounter / tSpan.TotalSeconds);
            fRxBw = (long)(lRxBytesCounter / tSpan.TotalSeconds);

            if (StatsChanged == null)
                return;

            StatsChanged(this, fTxBw, fRxBw);

            lTxBytesCounter = 0;
            lRxBytesCounter = 0;

            tStart = DateTime.Now;
        }

        private long lRxBytesCounter = 0;
        void CyDeviceInterface_BytesReceived(object sender, int Device, int Channel, int
NumBytes)
        {
            if (NumInEp > 0)
            {
                int aux = Array.IndexOf(EpIn, sender);
                if (aux >= 0)
                {
                    lRxBytesCounter += NumBytes;
                    if (BytesReceived != null)
                    {
                        BytesReceived(this, -1, aux, NumBytes);
                    }
                }
            }
        }

        private long lTxBytesCounter = 0;
        void CyDeviceInterface_BytesSent(object sender, int Device, int Channel, int
NumBytes)
        {
            if (NumOutEp > 0)
            {
```

```
                int aux = Array.IndexOf(EpOut, sender);
                if (aux >= 0)
                {
                    lTxBytesCounter += NumBytes;
                    if (BytesSent != null)
                    {
                        BytesSent(this, -1, aux, NumBytes);
                    }
                }
            }
        }

        /// <summary>
        /// Suspends the current device interface threads
        /// </summary>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool Suspend()
        {
            bool retval = false;
            if (EpIn != null)
            {
                foreach (CyInEndpointInterface var in EpIn)
                {
                    retval = true;
                    var.Suspend();
                }
            }
            if (EpOut != null)
            {
                foreach (CyOutEndpointInterface var in EpOut)
                {
                    retval = true;
                    var.Suspend();
                }
            }
            Suspended = true;
            return retval;
        }

        /// <summary>
        /// Resumes the current device interface threads
        /// </summary>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool Resume()
        {
            bool retval = false;
            if (EpIn != null)
            {
                foreach (CyInEndpointInterface var in EpIn)
                {
                    retval = true;
                    var.Resume();
                }
            }
            if (EpOut != null)
            {
                foreach (CyOutEndpointInterface var in EpOut)
                {
                    retval = true;
                    var.Resume();
                }
            }
```

```
            Suspended = false;
            return retval;
        }

        /// <summary>
        /// Stops the current device interface threads
        /// </summary>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool Stop()
        {
            bool retval = false;
            if (EpIn != null)
            {
                foreach (CyInEndpointInterface var in EpIn)
                {
                    retval = true;
                    var.Stop((int)(2 * var.QueueTo));
                }
            }
            if (EpOut != null)
            {
                foreach (CyOutEndpointInterface var in EpOut)
                {
                    retval = true;
                    var.Stop((int)(2 * var.QueueTo));
                }
            }
            return retval;
        }

        /// <summary>
        /// Stops the threads and releases all resources
        /// </summary>
        public void Dispose()
        {
            foreach (CyInEndpointInterface var in EpIn)
            {
                var.Dispose();
            }
            foreach (CyOutEndpointInterface var in EpOut)
            {
                var.Dispose();
            }
        }
    }
```

5.4 CyOutEndpoint Interface

```
    /// <summary>
    /// Creates a object wich interfaces with the 'Out' endpoint
    /// </summary>
    public class CyOutEndpointInterface
    {
        public event BytesTransferredEvent BytesSent;
        public event EpTerminated WorkFinished;

        private uint QUEUESZ;
        private uint BUFSZ;
        private uint QUEUETO;

        /// <summary>
        /// Queue depth
        /// </summary>
```

```
        public uint QueueSz
        {
            get
            {
                return QUEUESZ;
            }
        }

        /// <summary>
        /// Queue width
        /// </summary>
        public uint QueueWd
        {
            get
            {
                return BUFSZ;
            }
        }

        /// <summary>
        /// Queue timeout
        /// </summary>
        public uint QueueTo
        {
            get
            {
                return QUEUETO;
            }
        }

        TxWorker Worker;
        Thread tTx;

        Queue TxQueue;
        CyFX2Device AssociatedDevice;

        private bool bwRunning = false;

        /// <summary>
        /// Thread busy state
        /// </summary>
        public bool bwBusy
        {
            get
            {
                return bwRunning;
            }
        }

        public CyOutEndpointInterface()
        {
        }

        /// <summary>
        /// Suspends current thread
        /// </summary>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool Suspend()
        {
            Worker.Suspend();
            return true;
        }
```

```
        /// <summary>
        /// Resumes current thread
        /// </summary>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool Resume()
        {
            Worker.Resume();
            return true;
        }

        /// <summary>
        /// Stops current thread
        /// </summary>
        /// <param name="milisecondsTimeout">Timout in miliseconds</param>
        /// <returns>Returns true if thread stopped normally, returns false when a timout
occurs</returns>
        public bool Stop(int milisecondsTimeout)
        {
            bool retval = true;
            Worker.RequestStop();
            if (!tTx.Join(milisecondsTimeout))
            {
                retval = false;
                tTx.Abort();
            }
            return retval;
        }

        /// <summary>
        /// Stops current thread and releases resources;
        /// </summary>
        public void Dispose()
        {
            Stop((int)(2 * QueueTo));
        }

        /// <summary>
        /// Creates a new instance of the object to interface with the enpoint
        /// </summary>
        /// <param name="Device">Device associated with the object</param>
        /// <param name="EpQueue">Queue associated with the object</param>
        /// <param name="EpIndex">Endpoint index</param>
        /// <param name="QueueSz">Queue depth</param>
        /// <param name="QueueTo">Queue timeout</param>
        /// <param name="BufSz">Queue width</param>
        public CyOutEndpointInterface(CyFX2Device Device, Queue EpQueue, int EpIndex, uint
QueueSz, uint QueueTo, uint BufSz)
        {
            if (QueueSz == 0)
            {
                throw new Exception("Queue depth cannot be zero");
            }
            if (QueueTo == 0)
            {
                throw new Exception("Queue timeout cannot be zero");
            }
            if (BufSz == 0)
            {
                throw new Exception("Queue width cannot be zero");
            }
            QUEUESZ = QueueSz;
            BUFSZ = BufSz;
            QUEUETO = QueueTo;
```

```
            int firstEp = 0;
            if (Device == null)
            {
                throw new Exception("Cannot create object to a null CyFX2Device");
            }
            if (EpQueue == null)
            {
                throw new Exception("Cannot create object to a null Queue");
            }
            while ((!(Device.EndPoints[firstEp] is CyBulkEndPoint)) ||
(Device.EndPoints[firstEp].bIn))
            {
                firstEp++;
            }
            if (Device.EndPoints[EpIndex + firstEp].bIn)
            {
                throw new Exception("Endpoint cannot be of type 'Out'");
            }
            if (!(Device.EndPoints[EpIndex + firstEp] is CyBulkEndPoint))
            {
                throw new Exception("Endpoint must be Bulk");
            }

            TxQueue = EpQueue;
            BwParameter bwParams = new BwParameter();
            bwParams.BulkEndpoint = (CyBulkEndPoint)Device.EndPoints[EpIndex + firstEp];
            bwParams.EpIndex = EpIndex;
            bwParams.Device = Device;

            AssociatedDevice = Device;

            Worker = new TxWorker(bwParams, QUEUESZ, BUFSZ, QUEUETO, TxQueue);
            Worker.BytesSent += new BytesTransferredEvent(Worker_BytesSent);
            Worker.WorkFinished += new EpTerminated(Worker_WorkFinished);
            tTx = new Thread(new ThreadStart(Worker.DoWorkSync), ((int)(4 * (QUEUESZ *
BUFSZ))) + 131072);
            tTx.IsBackground = true;
            tTx.Start();
            bwRunning = true;
        }

        void Worker_WorkFinished(object sender, bool In, int Device, int Channel)
        {
            bwRunning = false;
            WorkFinished(this, false, -1, -1);
        }

        IAsyncResult res;
        bool bEventIsBusy = false;
        void Worker_BytesSent(object sender, int Device, int Channel, int NumBytes)
        {
            if (bEventIsBusy)
            {
                while (!res.IsCompleted)
                {
                    Thread.Sleep(0);
                }
            }
            bEventIsBusy = true;
            res = BytesSent.BeginInvoke(this, -1, -1, NumBytes, new
AsyncCallback(EventHandled), null);
        }
```

```
        void EventHandled(IAsyncResult itfAR)
        {
            bEventIsBusy = false;
            res = itfAR;
        }

        private class TxWorker
        {
            bool bTreadRepeat, bSuspend;
            BwParameter Parameters;
            Queue TxQueue;
            uint QUEUESZ;
            uint BUFSZ;
            uint QUEUETO;

            public event BytesTransferredEvent BytesSent;
            public event EpTerminated WorkFinished;

            public TxWorker(BwParameter parameters, uint QueueSz, uint BufSz, uint QueueTo,
Queue DataQueue)
            {
                Parameters = new BwParameter(parameters);
                QUEUESZ = QueueSz;
                BUFSZ = BufSz;
                QUEUETO = QueueTo;
                TxQueue = DataQueue;
            }

            public void DoWorkSync()
            {
                bTreadRepeat = true;
                bSuspend = false;
                CyUSBDevice selectedDevice = Parameters.Device;
                CyBulkEndPoint OutEndpt = Parameters.BulkEndpoint;

                OutEndpt.XferSize = (int)BUFSZ;
                OutEndpt.TimeOut = QUEUETO;

                byte[] auxBuf = new byte[BUFSZ], auxDequeue;
                int bytesToSend, bytesSent, bytesRemaining;

                while (bTreadRepeat)
                {
                    while (bSuspend)
                    {
                        Thread.Sleep(1);
                    }
                    if (TxQueue.Count > 0)
                    {
                        bytesToSend = 0;
                        do
                        {
                            lock (TxQueue.SyncRoot)
                            {
                                auxDequeue = (byte[])TxQueue.Dequeue();
                            }
                            if (auxDequeue != null)
                            {
                                if (auxDequeue.Length > 0)
                                {
                                    bytesToSend += auxDequeue.Length;
                                    Array.Resize<byte>(ref auxBuf, bytesToSend);
```

```
                                Array.Copy(auxDequeue, 0, auxBuf, bytesToSend -
auxDequeue.Length, auxDequeue.Length);
                            }
                        }
                    } while ((bytesToSend < BUFSZ) && (TxQueue.Count > 0));

                    if (bytesToSend > 0)
                    {

                        bytesRemaining = bytesToSend;
                        while (bytesRemaining > 0)
                        {
                            bytesSent = bytesRemaining;
                            if (OutEndpt.XferData(ref auxBuf, ref bytesSent))
                            {
                                if (bytesSent > 0)
                                {
                                    BytesSent(this, -1, -1, bytesSent);
                                }
                            }
                            else
                            {
                                Thread.Sleep(1);
                            }


                            bytesRemaining -= bytesSent;
                            if (bytesRemaining > 0)
                            {
                                Array.Reverse(auxBuf);
                                Array.Resize<byte>(ref auxBuf, bytesRemaining);
                                Array.Reverse(auxBuf);
                                bytesSent = bytesRemaining;
                            }
                        }
                    }
                    Thread.Sleep(1);
                }
                else
                {
                    Thread.Sleep(1);
                }
            }

            if (WorkFinished != null)
            {
                WorkFinished(this, true, -1, -1);
            }
            GC.KeepAlive(auxBuf);
        }

        public void DoWork()
        {
            bTreadRepeat = true;
            bSuspend = false;
            CyUSBDevice selectedDevice = Parameters.Device;
            CyBulkEndPoint OutEndpt = Parameters.BulkEndpoint;

            OutEndpt.XferSize = (int)BUFSZ;
            OutEndpt.TimeOut = QUEUETO;

            byte[][] cmdBufs = new byte[QUEUESZ][];

            ovLappedWrap[] ovLapped = new ovLappedWrap[QUEUESZ];
```

```
                byte[][] xferBufs = new byte[QUEUESZ][];
                int[] xferLens = new int[QUEUESZ];
                uint xferBufsUsed = 0, xferBufWrNext = 0, xferBufRdNext = 0;

                byte[] auxBuf = new byte[BUFSZ];

                byte[] remainingBuf = new byte[BUFSZ];
                uint remainingBufCount = 0;

                OVERLAPPED tmpOvLap = new OVERLAPPED();

                for (int i = 0; i < QUEUESZ; i++)
                {
                    cmdBufs[i] = new byte[CyConst.SINGLE_XFER_LEN];
                    xferBufs[i] = new byte[BUFSZ];
                    xferLens[i] = 0;
                    ovLapped[i] = new ovLappedWrap();
                    tmpOvLap.hEvent = (uint)PInvoke.CreateEvent(0, 0, 0, 0);
                    ovLapped[i].ovlap = tmpOvLap;

                }

                while (bTreadRepeat)
                {
                    while (bSuspend)
                    {
                        Thread.Sleep(1);
                    }
                    while (((TxQueue.Count > 0) || (remainingBufCount > 0)) &&
(xferBufsUsed < QUEUESZ))
                    {
                        while ((TxQueue.Count > 0) && (remainingBufCount < BUFSZ))
                        {
                            lock (TxQueue.SyncRoot)
                            {
                                auxBuf = (byte[])TxQueue.Dequeue();
                            }
                            Array.Resize<byte>(ref remainingBuf, (int)(remainingBufCount +
auxBuf.Length));
                            Array.Copy(auxBuf, 0, remainingBuf, remainingBufCount,
auxBuf.Length);
                            remainingBufCount += (uint)auxBuf.Length;
                        }
                        if (remainingBufCount > 0)
                        {
                            if (remainingBufCount > BUFSZ)
                            {
                                Array.Copy(remainingBuf, remainingBuf.Length -
remainingBufCount, xferBufs[xferBufWrNext], 0, BUFSZ);
                                xferLens[xferBufWrNext] = (int)BUFSZ;
                                remainingBufCount -= BUFSZ;
                            }
                            else
                            {
                                Array.Copy(remainingBuf, remainingBuf.Length -
remainingBufCount, xferBufs[xferBufWrNext], 0, remainingBufCount);
                                xferLens[xferBufWrNext] = (int)remainingBufCount;
                                remainingBufCount = 0;
                            }
                            while (cmdBufs == null)
                            {
                                Thread.Sleep(0);
                            }
```

```
                        while (cmdBufs[xferBufWrNext] == null)
                        {
                            Thread.Sleep(0);
                        }
                        while (xferBufs == null)
                        {
                            Thread.Sleep(0);
                        }
                        while (xferBufs[xferBufWrNext] == null)
                        {
                            Thread.Sleep(0);
                        }
                        while (xferLens == null)
                        {
                            Thread.Sleep(0);
                        }
                        while (ovLapped == null)
                        {
                            Thread.Sleep(0);
                        }
                        while (ovLapped[xferBufWrNext] == null)
                        {
                            Thread.Sleep(0);
                        }
                        OutEndpt.BeginDataXfer(ref cmdBufs[xferBufWrNext], ref
xferBufs[xferBufWrNext],
                                               ref xferLens[xferBufWrNext], ref
ovLapped[xferBufWrNext].Raw);
                        xferBufWrNext++;
                        xferBufWrNext %= QUEUESZ;
                        xferBufsUsed++;
                    }
                }
                if (xferBufsUsed > 0)
                {
                    while (ovLapped == null)
                    {
                        Thread.Sleep(0);
                    }
                    while (ovLapped[xferBufRdNext] == null)
                    {
                        Thread.Sleep(0);
                    }
                    tmpOvLap = ovLapped[xferBufRdNext].ovlap;
                    if (!OutEndpt.WaitForXfer(tmpOvLap.hEvent, QUEUETO))
                    {
                        if (!bTreadRepeat)
                        {
                            if (WorkFinished != null)
                            {
                                WorkFinished(this, false, -1, -1);
                            }
                            return;
                        }
                        uint u;
                        OutEndpt.Abort();
                        tmpOvLap = ovLapped[xferBufRdNext].ovlap;
                        u = PInvoke.WaitForSingleObject(tmpOvLap.hEvent, 1000)
                        if (u != 0)
                        {
                            if (!bTreadRepeat)
                            {
                                if (WorkFinished != null)
                                {
```

```
                                    WorkFinished(this, false, -1, -1);
                                }
                                return;
                            }
                        }
                    }
                    while (cmdBufs == null)
                    {
                        Thread.Sleep(0);
                    }
                    while (cmdBufs[xferBufRdNext] == null)
                    {
                        Thread.Sleep(0);
                    }
                    while (xferBufs == null)
                    {
                        Thread.Sleep(0);
                    }
                    while (xferBufs[xferBufRdNext] == null)
                    {
                        Thread.Sleep(0);
                    }
                    while (xferLens == null)
                    {
                        Thread.Sleep(0);
                    }
                    while (ovLapped == null)
                    {
                        Thread.Sleep(0);
                    }
                    while (ovLapped[xferBufRdNext] == null)
                    {
                        Thread.Sleep(0);
                    }
                    if (OutEndpt.FinishDataXfer(ref cmdBufs[xferBufRdNext], ref
xferBufs[xferBufRdNext],
                                                ref xferLens[xferBufRdNext], ref
ovLapped[xferBufRdNext].Raw))
                    {
                        BytesSent(this, -1, -1, xferLens[xferBufRdNext]);
                        xferLens[xferBufRdNext] = 0;
                        xferBufRdNext++;
                        xferBufRdNext %= QUEUESZ;
                        xferBufsUsed--;
                    }
                    else
                    {
                        xferBufRdNext++;
                        xferBufRdNext %= QUEUESZ;
                        xferBufsUsed--;
                    }
                }
                Thread.Sleep(0);
            }
            if (WorkFinished != null)
            {
                WorkFinished(this, false, -1, -1);
            }
            GC.KeepAlive(cmdBufs);
            GC.KeepAlive(xferBufs);
            GC.KeepAlive(ovLapped);
            GC.KeepAlive(auxBuf);
            GC.KeepAlive(tmpOvLap);
        }
```

```
        public void RequestStop()
        {
            bTreadRepeat = false;
        }

        public void Suspend()
        {
            bSuspend = true;
        }

        public void Resume()
        {
            bSuspend = false;
        }
    }
}

/// <summary>
/// Creates a object wich interfaces with the 'Out' endpoint
/// </summary>
public class CyInEndpointInterface
{
    public event BytesTransferredEvent BytesReceived;
    public event EpTerminated WorkFinished;

    private uint QUEUESZ;
    private uint BUFSZ;
    private uint QUEUETO;

    /// <summary>
    /// Queue depth
    /// </summary>
    public uint QueueSz
    {
        get
        {
            return QUEUESZ;
        }
    }

    /// <summary>
    /// Queue width
    /// </summary>
    public uint QueueWd
    {
        get
        {
            return BUFSZ;
        }
    }

    /// <summary>
    /// Queue timeout
    /// </summary>
    public uint QueueTo
    {
        get
        {
            return QUEUETO;
        }
    }

    RxWorker Worker;
```

```
        Thread tRx;

        Queue RxQueue;
        CyFX2Device AssociatedDevice;

        private bool bwRunning = false;

        /// <summary>
        /// Thread busy state
        /// </summary>
        public bool bwBusy
        {
            get
            {
                return bwRunning;
            }
        }

        public CyInEndpointInterface()
        {
        }

        /// <summary>
        /// Suspends current thread
        /// </summary>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool Suspend()
        {
            Worker.Suspend();
            return true;
        }

        /// <summary>
        /// Resumes current thread
        /// </summary>
        /// <returns>Returns true if operation completes succefully, otherwise
false</returns>
        public bool Resume()
        {
            Worker.Resume();
            return true;
        }

        /// <summary>
        /// Stops current thread
        /// </summary>
        /// <param name="milisecondsTimeout">Timout in miliseconds</param>
        /// <returns>Returns true if thread stopped normally, returns false when a timout
occurs</returns>
        public bool Stop(int milisecondsTimeout)
        {
            Worker.RequestStop();
            if (!tRx.Join(milisecondsTimeout))
            {
                tRx.Abort();
            }
            return true;
        }

        /// <summary>
        /// Stops current thread and releases resources;
        /// </summary>
        public void Dispose()
```

```
        {
            Stop((int)(2 * QueueTo));
        }

        /// <summary>
        /// Creates a new instance of the object to interface with the enpoint
        /// </summary>
        /// <param name="Device">Device associated with the object</param>
        /// <param name="EpQueue">Queue associated with the object</param>
        /// <param name="EpIndex">Endpoint index</param>
        /// <param name="QueueSz">Queue depth</param>
        /// <param name="QueueTo">Queue timeout</param>
        /// <param name="BufSz">Queue width</param>
        public CyInEndpointInterface(CyFX2Device Device, Queue EpQueue, int EpIndex, uint
QueueSz, uint QueueTo, uint BufSz)
        {
            if (QueueSz == 0)
            {
                throw new Exception("Queue depth cannot be zero");
            }
            if (QueueTo == 0)
            {
                throw new Exception("Queue timeout cannot be zero");
            }
            if (BufSz == 0)
            {
                throw new Exception("Queue width cannot be zero");
            }
            QUEUESZ = QueueSz;
            BUFSZ = BufSz;
            QUEUETO = QueueTo;

            int firstEp = 0;
            if (Device == null)
            {
                throw new Exception("Cannot create object to a null CyFX2Device");
            }
            if (EpQueue == null)
            {
                throw new Exception("Cannot create object to a null Queue");
            }
            while ((!(Device.EndPoints[firstEp] is CyBulkEndPoint)) ||
(!Device.EndPoints[firstEp].bIn))
            {
                firstEp++;
            }
            if (!(Device.EndPoints[EpIndex + firstEp] is CyBulkEndPoint))
            {
                throw new Exception("Endpoint must be Bulk");
            }
            if (!Device.EndPoints[EpIndex + firstEp].bIn)
            {
                throw new Exception("Endpoint cannot be of type 'Out'");
            }

            RxQueue = EpQueue;
            BwParameter bwParams = new BwParameter();
            bwParams.BulkEndpoint = (CyBulkEndPoint)Device.EndPoints[EpIndex + firstEp];
            bwParams.EpIndex = EpIndex;
            bwParams.Device = Device;

            AssociatedDevice = Device;

            Worker = new RxWorker(bwParams, QUEUESZ, BUFSZ, QUEUETO, RxQueue);
```

```
            Worker.BytesReceived += new BytesTransferredEvent(Worker_BytesReceived);
            Worker.WorkFinished += new EpTerminated(Worker_WorkFinished);
            tRx = new Thread(new ThreadStart(Worker.DoWorkSync), ((int)(4 * (QUEUESZ *
BUFSZ))) + 131072);
            tRx.IsBackground = true;
            tRx.Start();
            bwRunning = true;

        }

        void Worker_WorkFinished(object sender, bool In, int Device, int Channel)
        {
            bwRunning = false;
            WorkFinished(this, true, -1, -1);
        }

        IAsyncResult res;
        bool bEventIsBusy = false;
        void Worker_BytesReceived(object sender, int Device, int Channel, int NumBytes)
        {
            if (bEventIsBusy)
            {
                while (!res.IsCompleted)
                {
                    Thread.Sleep(0);
                }
            }
            bEventIsBusy = true;
            res = BytesReceived.BeginInvoke(this, -1, -1, NumBytes, new
AsyncCallback(EventHandled), null);
        }

        void EventHandled(IAsyncResult itfAR)
        {
            bEventIsBusy = false;
            res = itfAR;
        }

        private class RxWorker
        {
            bool bTreadRepeat, bSuspend;
            BwParameter Parameters;
            Queue RxQueue;
            uint QUEUESZ;
            uint BUFSZ;
            uint QUEUETO;

            public event BytesTransferredEvent BytesReceived;
            public event EpTerminated WorkFinished;

            public RxWorker(BwParameter parameters, uint QueueSz, uint BufSz, uint QueueTo,
Queue DataQueue)
            {
                Parameters = new BwParameter(parameters);
                QUEUESZ = QueueSz;
                BUFSZ = BufSz;
                QUEUETO = QueueTo;
                RxQueue = DataQueue;
            }

            byte[] temp;
            int abc;

            public void DoWorkSync()
```

```
{
    bTreadRepeat = true;
    bSuspend = false;
    CyUSBDevice selectedDevice = Parameters.Device;
    CyBulkEndPoint InEndpt = Parameters.BulkEndpoint;

    InEndpt.XferSize = (int)BUFSZ;
    InEndpt.TimeOut = QUEUETO;

    byte[] auxBuf = new byte[BUFSZ];
    byte[] bufToEnqueue = null;
    int bytesToReceive;

    while (bTreadRepeat)
    {
        while (bSuspend)
        {
            Thread.Sleep(1);
        }
        if (RxQueue.Count < QUEUESZ)
        {
            bytesToReceive = (int)BUFSZ;
            if (InEndpt.XferData(ref auxBuf, ref bytesToReceive))
            {
                temp = (byte[])auxBuf.Clone();
                abc = bytesToReceive;
                if (bytesToReceive > 0)
                {
                    bufToEnqueue = new byte[bytesToReceive];
                    Array.Copy(auxBuf, bufToEnqueue, bytesToReceive);
                    lock (RxQueue.SyncRoot)
                    {
                        RxQueue.Enqueue(bufToEnqueue);
                    }
                    BytesReceived(this, -1, -1, bytesToReceive);
                    Thread.Sleep(0);
                }
                else
                {
                    Thread.Sleep(1);
                }
            }
            else
            {
                if (bytesToReceive > 0)
                {
                }
                Thread.Sleep(1);
            }
        }
    }

    if (WorkFinished != null)
    {
        WorkFinished(this, true, -1, -1);
    }
    GC.KeepAlive(auxBuf);
    GC.KeepAlive(bufToEnqueue);
}

public void DoWork()
{
    bTreadRepeat = true;
    bSuspend = false;
```

```
CyUSBDevice selectedDevice = Parameters.Device;
CyBulkEndPoint InEndpt = Parameters.BulkEndpoint;

InEndpt.XferSize = (int)BUFSZ;
InEndpt.TimeOut = QUEUETO;

byte[][] cmdBufs = new byte[QUEUESZ][];

ovLappedWrap[] ovLapped = new ovLappedWrap[QUEUESZ];

byte[][] xferBufs = new byte[QUEUESZ][];
uint xferBufRdNext = 0;
int receivedBytes = 0, bufferSize = 0;

byte[] auxBuf = new byte[BUFSZ];

uint u;

OVERLAPPED tmpOvLap = new OVERLAPPED();

bufferSize = (int)BUFSZ;
for (int i = 0; i < QUEUESZ; i++)
{
    cmdBufs[i] = new byte[CyConst.SINGLE_XFER_LEN];
    xferBufs[i] = new byte[BUFSZ];
    ovLapped[i] = new ovLappedWrap();

    tmpOvLap.hEvent = (uint)PInvoke.CreateEvent(0, 0, 0, 0);
    ovLapped[i].ovlap = tmpOvLap;
    InEndpt.BeginDataXfer(ref cmdBufs[i], ref xferBufs[i],
                          ref bufferSize, ref ovLapped[i].Raw);
}

while (bTreadRepeat)
{
    while (bSuspend)
    {
        Thread.Sleep(1);
    }
    if (RxQueue.Count < QUEUESZ)
    {
        tmpOvLap = ovLapped[xferBufRdNext].ovlap;
        if (!InEndpt.WaitForXfer(tmpOvLap.hEvent, QUEUETO))
        {
            if (!bTreadRepeat)
            {
                if (WorkFinished != null)
                {
                    WorkFinished(this, false, -1, -1);
                }
                return;
            }
            InEndpt.Abort();
            while (ovLapped == null)
            {
                Thread.Sleep(0);
            }
            while (ovLapped[xferBufRdNext] == null)
            {
                Thread.Sleep(0);
            }
            tmpOvLap = ovLapped[xferBufRdNext].ovlap;
            u = PInvoke.WaitForSingleObject(tmpOvLap.hEvent, 1000);
            if (u != 0)
```

```
            {
                if (!bTreadRepeat)
                {
                    if (WorkFinished != null)
                    {
                        WorkFinished(this, false, -1, -1);
                    }
                    return;
                }
            }
        }

        while (cmdBufs == null)
        {
            Thread.Sleep(0);
        }
        while (cmdBufs[xferBufRdNext] == null)
        {
            Thread.Sleep(0);
        }
        while (xferBufs == null)
        {
            Thread.Sleep(0);
        }
        while (xferBufs[xferBufRdNext] == null)
        {
            Thread.Sleep(0);
        }
        while (ovLapped == null)
        {
            Thread.Sleep(0);
        }
        while (ovLapped[xferBufRdNext] == null)
        {
            Thread.Sleep(0);
        }
        receivedBytes = (int)BUFSZ;
        if (InEndpt.FinishDataXfer(ref cmdBufs[xferBufRdNext], ref
xferBufs[xferBufRdNext],
                                   ref receivedBytes, ref
ovLapped[xferBufRdNext].Raw))
        {
            if (receivedBytes > 0)
            {
                Array.Resize<byte>(ref auxBuf, receivedBytes);
                if (xferBufs[xferBufRdNext] != null)
                {
                    Array.Copy(xferBufs[xferBufRdNext], auxBuf,
receivedBytes);

                    lock (RxQueue.SyncRoot)
                    {
                        RxQueue.Enqueue((byte[])auxBuf.Clone());
                    }
                    BytesReceived(this, -1, -1, receivedBytes);
                }
                else
                {
                    xferBufs[xferBufRdNext] = new byte[(int)BUFSZ];
                }
            }
        }
        while (cmdBufs == null)
        {
            Thread.Sleep(0);
```

```
                        }
                        while (cmdBufs[xferBufRdNext] == null)
                        {
                            Thread.Sleep(0);
                        }
                        while (xferBufs == null)
                        {
                            Thread.Sleep(0);
                        }
                        while (xferBufs[xferBufRdNext] == null)
                        {
                            Thread.Sleep(0);
                        }
                        while (ovLapped == null)
                        {
                            Thread.Sleep(0);
                        }
                        while (ovLapped[xferBufRdNext] == null)
                        {
                            Thread.Sleep(0);
                        }
                        bufferSize = (int)BUFSZ;
                        InEndpt.BeginDataXfer(ref cmdBufs[xferBufRdNext], ref
xferBufs[xferBufRdNext],
                                              ref bufferSize, ref
ovLapped[xferBufRdNext].Raw);
                        xferBufRdNext++;
                        xferBufRdNext %= QUEUESZ;
                    }
                    Thread.Sleep(0);
                }
                if (WorkFinished != null)
                {
                    WorkFinished(this, true, -1, -1);
                }
                GC.KeepAlive(cmdBufs);
                GC.KeepAlive(xferBufs);
                GC.KeepAlive(ovLapped);
                GC.KeepAlive(auxBuf);
                GC.KeepAlive(tmpOvLap);
            }

            public void RequestStop()
            {
                bTreadRepeat = false;
            }

            public void Suspend()
            {
                bSuspend = true;
            }

            public void Resume()
            {
                bSuspend = false;
            }
        }
    }

    class BwParameter
    {
        public CyFX2Device Device;
        public CyBulkEndPoint BulkEndpoint;
        public int EpIndex;
```

```
        public BwParameter()
        {
            Device = null;
            BulkEndpoint = null;
            EpIndex = 0;
        }

        public BwParameter(BwParameter c)
        {
            Device = c.Device;
            BulkEndpoint = c.BulkEndpoint;
            EpIndex = c.EpIndex;
        }
    }

    class ovLappedWrap
    {
        public byte[] Raw;

        public ovLappedWrap()
        {
            Raw = new byte[20];
        }

        public unsafe OVERLAPPED ovlap
        {
            get
            {

                lock (this)
                {
                    fixed (byte* tmp = Raw)
                    {
                        OVERLAPPED* auxptr = (OVERLAPPED*)tmp;
                        return *auxptr;
                    }
                }
            }
            set
            {
                lock (this)
                {
                OVERLAPPED* tmp = &value;
                {
                    byte* auxptr = (byte*)tmp;
                    for (int i = 0; i < 20; i++)
                    {
                        Raw[i] = *(auxptr + i);
                    }
                }
                }
            }
        }
    }

}
```